Документация к Postgres Pro Standard 10.23.1



ООО «Постгрес Профессиональный»
https://postgrespro.ru

Документация к Postgres Pro Standard 10.23.1
ООО «Постгрес Профессиональный»
Авторские права © 2016–2022 Постгрес Профессиональный

Юридическое уведомление

Представленная документация предназначена исключительно для использования с программным обеспечением «СУБД Postgres Pro» и для пользователей данной СУБД.

Не допускается использование документации для сторонних продуктов или в составе документации к иным продуктам.

Иные условия использования данной документации приведены в Пользовательском соглашении.

Postgres Pro © 2016–2022 — Общество с ограниченной ответственностью «Постгрес Профессиональный».

Общество с ограниченной ответственностью «Постгрес Профессиональный» (Postgres Professional) не несёт никакой ответственности за любые повреждения, включая потерю дохода, нанесённые прямым или непрямым, специальным или случайным использованием программного обеспечения «СУБД Postgres Pro» во всех версиях и/или его документации, даже если общество с ограниченной ответственностью «Постгрес Профессиональный» было извещено о возможности таких повреждений.

Общество с ограниченной ответственностью «Постгрес Профессиональный» (Postgres Professional) специально отказывается предоставлять любые гарантии, включая, но не ограничиваясь только этими гарантиями: неявные гарантии пригодности товара или пригодности для отдельной цели. Программное обеспечение «СУБД Postgres Pro» во всех версиях и/или документация к нему предоставляется на основе принципа «как есть», и общество с ограниченной ответственностью «Постгрес Профессиональный» не обязано предоставлять сопровождение, поддержку, обновления, расширения или изменения.

Для распространения этой документации или её частей требуется явное письменное разрешение от Общества с ограниченной ответственностью «Постгрес Профессиональный».

Документация программного обеспечения «СУБД Postgres Pro» во всех версиях основана на документации PostgreSQL, которая распространяется по следующей лицензии:

PostgreSQL © 1996–2022 — PostgreSQL Global Development Group.

Postgres95 © 1994–5 — Регенты университета Калифорнии.

Настоящим разрешается использование, копирование, модификация и распространение данного программного обеспечения и документации для любых целей, бесплатно и без письменного разрешения, при условии сохранения во всех копиях приведённого выше уведомления об авторских правах и данного параграфа вместе с двумя последующими параграфами.

УНИВЕРСИТЕТ КАЛИФОРНИИ НИ В КОЕЙ МЕРЕ НЕ НЕСЁТ ОТВЕТСТВЕННОСТИ ЗА ПРЯМОЙ, КОСВЕННЫЙ, НАМЕРЕННЫЙ, СЛУЧАЙНЫЙ ИЛИ СПРОВОЦИРОВАННЫЙ УЩЕРБ, В ТОМ ЧИСЛЕ ПОТЕРЯННЫЕ ПРИБЫЛИ, СВЯЗАННЫЙ С ИСПОЛЬЗОВАНИЕМ ЭТОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И ЕГО ДОКУМЕНТАЦИИ, ДАЖЕ ЕСЛИ УНИВЕРСИТЕТ КАЛИФОРНИИ БЫЛ УВЕДОМЛЁН О ВОЗМОЖНОСТИ ТАКОГО УЩЕРБА.

УНИВЕРСИТЕТ КАЛИФОРНИИ ЯВНЫМ ОБРАЗОМ ОТКАЗЫВАЕТСЯ ОТ ЛЮБЫХ ГАРАНТИЙ, В ЧАСТНОСТИ ПОДРАЗУМЕВАЕМЫХ ГАРАНТИЙ КОММЕРЧЕСКОЙ ВЫГОДЫ ИЛИ ПРИГОДНОСТИ ДЛЯ КАКОЙ-ЛИБО ЦЕЛИ. НАСТОЯЩЕЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ПРЕДОСТАВЛЯЕТСЯ В ВИДЕ «КАК ЕСТЬ», И УНИВЕРСИТЕТ КАЛИФОРНИИ НЕ ДАЁТ НИКАКИХ ОБЯЗАТЕЛЬСТВ ПО ЕГО ОБСЛУЖИВАНИЮ, ПОДДЕРЖКЕ, ОБНОВЛЕНИЮ, УЛУЧШЕНИЮ ИЛИ МОДИФИКАЦИИ.

Предисловие xxi
1. Что такое Postgres Pro Standard? xxi
2. Различия между Postgres Pro Standard и PostgreSQL xxi
3. Краткая история PostgreSQL xxiii
3.1. Проект POSTGRES в Беркли xxiii
3.2. Postgres95 xxiii
3.3. PostgreSQL xxiv
4. Соглашения xxiv
5. Как правильно сообщить об ошибке xxiv
5.1. Диагностика ошибок xxv
5.2. Что сообщать xxv
5.3. Куда сообщать xxvi
I. Введение 1
1. Начало 2
1.1. Установка 2
1.2. Основы архитектуры 2
1.3. Создание базы данных 2
1.4. Подключение к базе данных 4
2. Язык SQL 6
2.1. Введение 6
2.2. Основные понятия 6
2.3. Создание таблицы 6
2.4. Добавление строк в таблицу 7
2.5. Выполнение запроса 8
2.6. Соединения таблиц 9
2.7. Агрегатные функции 11
2.8. Изменение данных 13
2.9. Удаление данных 13
3. Расширенные возможности 14
3.1. Введение 14
3.2. Представления 14
3.3. Внешние ключи 14
3.4. Транзакции 15
3.5. Оконные функции 17
3.6. Наследование 19
3.7. Заключение 21
II. Язык SQL 22
4. Синтаксис SQL 23
4.1. Лексическая структура 23
4.2. Выражения значения 32
4.3. Вызов функций 44
5. Определение данных 47
5.1. Основы таблиц 47
5.2. Значения по умолчанию 48
5.3. Ограничения 49
5.4. Системные столбцы 57
5.5. Изменение таблиц 58
5.6. Права 60
5.7. Политики защиты строк 61
5.8. Схемы 67
5.9. Наследование 71
5.10. Секционирование таблиц 75
5.11. Сторонние данные 86
5.12. Другие объекты баз данных 87
5.13. Отслеживание зависимостей 87
6. Модификация данных 89
6.1. Добавление данных 89
6.2. Изменение данных 90

6.3. Удаление данных ... 91
6.4. Возврат данных из изменённых строк ... 91
7. Запросы ... 93
7.1. Обзор ... 93
7.2. Табличные выражения ... 93
7.3. Списки выборки ... 107
7.4. Сочетание запросов ... 109
7.5. Сортировка строк ... 109
7.6. LIMIT и OFFSET ... 110
7.7. Списки VALUES ... 111
7.8. Запросы WITH (Общие табличные выражения) ... 112
8. Типы данных ... 118
8.1. Числовые типы ... 120
8.2. Денежные типы ... 124
8.3. Символьные типы ... 125
8.4. Двоичные типы данных ... 127
8.5. Типы даты/времени ... 129
8.6. Логический тип ... 139
8.7. Типы перечислений ... 140
8.8. Геометрические типы ... 141
8.9. Типы, описывающие сетевые адреса ... 144
8.10. Битовые строки ... 146
8.11. Типы, предназначенные для текстового поиска ... 147
8.12. Тип UUID ... 149
8.13. Тип XML ... 150
8.14. Типы JSON ... 152
8.15. Массивы ... 158
8.16. Составные типы ... 167
8.17. Диапазонные типы ... 172
8.18. Идентификаторы объектов ... 178
8.19. Тип pg_lsn ... 179
8.20. Псевдотипы ... 179
9. Функции и операторы ... 182
9.1. Логические операторы ... 182
9.2. Функции и операторы сравнения ... 182
9.3. Математические функции и операторы ... 185
9.4. Строковые функции и операторы ... 189
9.5. Функции и операторы двоичных строк ... 205
9.6. Функции и операторы для работы с битовыми строками ... 207
9.7. Поиск по шаблону ... 208
9.8. Функции форматирования данных ... 224
9.9. Операторы и функции даты/времени ... 231
9.10. Функции для перечислений ... 244
9.11. Геометрические функции и операторы ... 245
9.12. Функции и операторы для работы с сетевыми адресами ... 249
9.13. Функции и операторы текстового поиска ... 251
9.14. XML-функции ... 257
9.15. Функции и операторы JSON ... 269
9.16. Функции для работы с последовательностями ... 279
9.17. Условные выражения ... 281
9.18. Функции и операторы для работы с массивами ... 284
9.19. Диапазонные функции и операторы ... 288
9.20. Агрегатные функции ... 290
9.21. Оконные функции ... 298
9.22. Выражения подзапросов ... 300
9.23. Сравнение табличных строк и массивов ... 303
9.24. Функции, возвращающие множества ... 306
9.25. Системные информационные функции ... 309

9.26. Функции для системного администрирования ... 327
9.27. Триггерные функции ... 346
9.28. Функции событийных триггеров ... 346
10. Преобразование типов ... 350
10.1. Обзор ... 350
10.2. Операторы ... 351
10.3. Функции ... 355
10.4. Хранимое значение ... 359
10.5. UNION, CASE и связанные конструкции ... 359
10.6. Выходные столбцы SELECT ... 361
11. Индексы ... 362
11.1. Введение ... 362
11.2. Типы индексов ... 363
11.3. Составные индексы ... 365
11.4. Индексы и предложения ORDER BY ... 366
11.5. Объединение нескольких индексов ... 367
11.6. Уникальные индексы ... 367
11.7. Индексы по выражениям ... 368
11.8. Частичные индексы ... 368
11.9. Семейства и классы операторов ... 371
11.10. Индексы и правила сортировки ... 372
11.11. Сканирование только индекса ... 373
11.12. Контроль использования индексов ... 375
12. Полнотекстовый поиск ... 377
12.1. Введение ... 377
12.2. Таблицы и индексы ... 381
12.3. Управление текстовым поиском ... 383
12.4. Дополнительные возможности ... 389
12.5. Анализаторы ... 394
12.6. Словари ... 395
12.7. Пример конфигурации ... 404
12.8. Тестирование и отладка текстового поиска ... 405
12.9. Типы индексов, предпочитаемые для текстового поиска ... 409
12.10. Поддержка psql ... 410
12.11. Ограничения ... 412
13. Управление конкурентным доступом ... 414
13.1. Введение ... 414
13.2. Изоляция транзакций ... 414
13.3. Явные блокировки ... 420
13.4. Проверки целостности данных на уровне приложения ... 426
13.5. Ограничения ... 428
13.6. Блокировки и индексы ... 428
14. Оптимизация производительности ... 430
14.1. Использование EXPLAIN ... 430
14.2. Статистика, используемая планировщиком ... 441
14.3. Управление планировщиком с помощью явных предложений JOIN ... 444
14.4. Наполнение базы данных ... 446
14.5. Оптимизация, угрожающая стабильности ... 449
15. Параллельный запрос ... 450
15.1. Как работают параллельно выполняемые запросы ... 450
15.2. Когда может применяться распараллеливание запросов? ... 451
15.3. Параллельные планы ... 452
15.4. Безопасность распараллеливания ... 454
III. Администрирование сервера ... 456
16. Установка из двоичных пакетов ... 457
16.1. Установка Postgres Pro Standard в Linux ... 457
16.2. Установка Postgres Pro Standard в Windows ... 464

16.3. Установка дополнительно поставляемых модулей 468
16.4. Миграция на Postgres Pro 468
17. Подготовка к работе и сопровождение сервера 469
17.1. Учётная запись пользователя Postgres Pro 469
17.2. Создание кластера баз данных 469
17.3. Запуск сервера баз данных 471
17.4. Управление ресурсами ядра 474
17.5. Выключение сервера 484
17.6. Обновление кластера Postgres Pro 485
17.7. Защита от подмены сервера 487
17.8. Возможности шифрования 488
17.9. Защита соединений TCP/IP с применением SSL 489
17.10. Защита соединений TCP/IP с применением туннелей SSH 492
17.11. Регистрация журнала событий в Windows 493
18. Настройка сервера 495
18.1. Изменение параметров 495
18.2. Расположения файлов 499
18.3. Подключения и аутентификация 500
18.4. Потребление ресурсов 505
18.5. Журнал предзаписи 512
18.6. Репликация 519
18.7. Планирование запросов 524
18.8. Регистрация ошибок и протоколирование работы сервера 530
18.9. Статистика времени выполнения 541
18.10. Автоматическая очистка 542
18.11. Параметры клиентских сеансов по умолчанию 544
18.12. Управление блокировками 553
18.13. Совместимость с разными версиями и платформами 554
18.14. Обработка ошибок 557
18.15. Предопределённые параметры 557
18.16. Внесистемные параметры 559
18.17. Параметры для разработчиков 559
18.18. Краткие аргументы 562
19. Аутентификация клиентского приложения 564
19.1. Файл `pg_hba.conf` 564
19.2. Файл сопоставления имён пользователей 571
19.3. Методы аутентификации 572
19.4. Проблемы аутентификации 582
20. Роли базы данных 583
20.1. Роли базы данных 583
20.2. Атрибуты ролей 584
20.3. Членство в роли 585
20.4. Удаление ролей 587
20.5. Предопределённые роли 587
20.6. Безопасность функций 588
21. Управление базами данных 590
21.1. Обзор 590
21.2. Создание базы данных 590
21.3. Шаблоны баз данных 591
21.4. Конфигурирование баз данных 592
21.5. Удаление базы данных 593
21.6. Табличные пространства 593
22. Локализация 596
22.1. Поддержка языковых стандартов 596
22.2. Поддержка правил сортировки 598
22.3. Поддержка кодировок 604
23. Регламентные задачи обслуживания базы данных 611
23.1. Регламентная очистка 611

23.2. Регулярная переиндексация ... 620
23.3. Обслуживание журнала ... 620
24. Резервное копирование и восстановление ... 623
24.1. Выгрузка в SQL ... 623
24.2. Резервное копирование на уровне файлов ... 626
24.3. Непрерывное архивирование и восстановление на момент времени (Point-in-Time Recovery, PITR) ... 627
25. Отказоустойчивость, балансировка нагрузки и репликация ... 640
25.1. Сравнение различных решений ... 640
25.2. Трансляция журналов на резервные серверы ... 644
25.3. Отработка отказа ... 653
25.4. Другие методы трансляции журнала ... 654
25.5. Горячий резерв ... 656
26. Конфигурация восстановления ... 664
26.1. Параметры восстановления из архива ... 664
26.2. Параметры управления восстановлением ... 665
26.3. Параметры резервного сервера ... 666
27. Мониторинг работы СУБД ... 668
27.1. Стандартные инструменты Unix ... 668
27.2. Сборщик статистики ... 669
27.3. Просмотр информации о блокировках ... 706
27.4. Отслеживание выполнения ... 706
27.5. Динамическая трассировка ... 708
28. Мониторинг использования диска ... 720
28.1. Определение использования диска ... 720
28.2. Ошибка переполнения диска ... 721
29. Надёжность и журнал предзаписи ... 722
29.1. Надёжность ... 722
29.2. Журнал предзаписи (WAL) ... 724
29.3. Асинхронное подтверждение транзакций ... 725
29.4. Настройка WAL ... 726
29.5. Внутреннее устройство WAL ... 729
30. Логическая репликация ... 731
30.1. Публикация ... 731
30.2. Подписка ... 732
30.3. Конфликты ... 733
30.4. Ограничения ... 734
30.5. Архитектура ... 734
30.6. Мониторинг ... 735
30.7. Безопасность ... 735
30.8. Параметры конфигурации ... 736
30.9. Быстрая настройка ... 736
IV. Клиентские интерфейсы ... 737
31. libpq — библиотека для языка C ... 738
31.1. Функции управления подключением к базе данных ... 738
31.2. Функции, описывающие текущее состояние подключения ... 751
31.3. Функции для исполнения команд ... 757
31.4. Асинхронная обработка команд ... 773
31.5. Построчное извлечение результатов запроса ... 777
31.6. Отмена запросов в процессе выполнения ... 778
31.7. Интерфейс быстрого пути ... 779
31.8. Асинхронное уведомление ... 780
31.9. Функции, связанные с командой COPY ... 781
31.10. Функции управления ... 785
31.11. Функции разного назначения ... 787
31.12. Обработка замечаний ... 790
31.13. Система событий ... 791
31.14. Переменные окружения ... 797

31.15. Файл паролей ... 798
31.16. Файл соединений служб ... 799
31.17. Получение параметров соединения через LDAP ... 800
31.18. Поддержка SSL ... 800
31.19. Поведение в многопоточных программах ... 805
31.20. Сборка программ с libpq ... 805
31.21. Примеры программ ... 807
32. Большие объекты ... 817
32.1. Введение ... 817
32.2. Особенности реализации ... 817
32.3. Клиентские интерфейсы ... 817
32.4. Серверные функции ... 821
32.5. Пример программы ... 822
33. ECPG — встраиваемый SQL в C ... 828
33.1. Концепция ... 828
33.2. Управление подключениями к базе данных ... 828
33.3. Запуск команд SQL ... 831
33.4. Использование переменных среды ... 833
33.5. Динамический SQL ... 846
33.6. Библиотека pgtypes ... 848
33.7. Использование областей дескрипторов ... 860
33.8. Обработка ошибок ... 872
33.9. Директивы препроцессора ... 878
33.10. Компиляция программ со встраиваемым SQL ... 880
33.11. Библиотечные функции ... 881
33.12. Большие объекты ... 882
33.13. Приложения на C++ ... 883
33.14. Команды встраиваемого SQL ... 887
33.15. Режим совместимости с Informix ... 908
33.16. Внутреннее устройство ... 922
34. Информационная схема ... 924
34.1. Схема ... 924
34.2. Типы данных ... 924
34.3. `information_schema_catalog_name` ... 925
34.4. `administrable_role_authorizations` ... 925
34.5. `applicable_roles` ... 925
34.6. `attributes` ... 926
34.7. `character_sets` ... 929
34.8. `check_constraint_routine_usage` ... 930
34.9. `check_constraints` ... 931
34.10. `collations` ... 931
34.11. `collation_character_set_applicability` ... 931
34.12. `column_domain_usage` ... 932
34.13. `column_options` ... 932
34.14. `column_privileges` ... 933
34.15. `column_udt_usage` ... 933
34.16. `columns` ... 934
34.17. `constraint_column_usage` ... 939
34.18. `constraint_table_usage` ... 940
34.19. `data_type_privileges` ... 940
34.20. `domain_constraints` ... 941
34.21. `domain_udt_usage` ... 941
34.22. `domains` ... 942
34.23. `element_types` ... 945
34.24. `enabled_roles` ... 947
34.25. `foreign_data_wrapper_options` ... 948
34.26. `foreign_data_wrappers` ... 948

34.27. foreign_server_options ... 948
34.28. foreign_servers ... 949
34.29. foreign_table_options ... 949
34.30. foreign_tables ... 950
34.31. key_column_usage ... 950
34.32. parameters ... 951
34.33. referential_constraints ... 953
34.34. role_column_grants ... 954
34.35. role_routine_grants ... 955
34.36. role_table_grants ... 955
34.37. role_udt_grants ... 956
34.38. role_usage_grants ... 956
34.39. routine_privileges ... 957
34.40. routines ... 958
34.41. schemata ... 964
34.42. sequences ... 964
34.43. sql_features ... 966
34.44. sql_implementation_info ... 966
34.45. sql_languages ... 967
34.46. sql_packages ... 967
34.47. sql_parts ... 968
34.48. sql_sizing ... 968
34.49. sql_sizing_profiles ... 969
34.50. table_constraints ... 969
34.51. table_privileges ... 970
34.52. tables ... 971
34.53. transforms ... 972
34.54. triggered_update_columns ... 972
34.55. triggers ... 973
34.56. udt_privileges ... 975
34.57. usage_privileges ... 975
34.58. user_defined_types ... 976
34.59. user_mapping_options ... 978
34.60. user_mappings ... 978
34.61. view_column_usage ... 979
34.62. view_routine_usage ... 979
34.63. view_table_usage ... 980
34.64. views ... 981
V. Серверное программирование ... 982
35. Расширение SQL ... 983
35.1. Как реализована расширяемость ... 983
35.2. Система типов Postgres Pro ... 983
35.3. Пользовательские функции ... 985
35.4. Функции на языке запросов (SQL) ... 985
35.5. Перегрузка функций ... 999
35.6. Категории изменчивости функций ... 1000
35.7. Функции на процедурных языках ... 1002
35.8. Внутренние функции ... 1002
35.9. Функции на языке C ... 1002
35.10. Пользовательские агрегатные функции ... 1023
35.11. Пользовательские типы ... 1030
35.12. Пользовательские операторы ... 1034
35.13. Информация для оптимизации операторов ... 1035
35.14. Интерфейсы расширений для индексов ... 1039
35.15. Упаковывание связанных объектов в расширение ... 1052
35.16. Инфраструктура сборки расширений ... 1061

36. Триггеры ... 1064
36.1. Обзор механизма работы триггеров ... 1064
36.2. Видимость изменений в данных ... 1067
36.3. Триггерные функции на языке C ... 1067
36.4. Полный пример триггера ... 1070
37. Триггеры событий ... 1073
37.1. Обзор механизма работы триггеров событий ... 1073
37.2. Матрица срабатывания триггеров событий ... 1074
37.3. Триггерные функции событий на языке C ... 1078
37.4. Полный пример триггера события ... 1079
37.5. Пример событийного триггера, обрабатывающего перезапись таблицы ... 1080
38. Система правил ... 1082
38.1. Дерево запроса ... 1082
38.2. Система правил и представления ... 1084
38.3. Материализованные представления ... 1090
38.4. Правила для INSERT, UPDATE и DELETE ... 1093
38.5. Правила и права ... 1103
38.6. Правила и статус команд ... 1105
38.7. Сравнение правил и триггеров ... 1105
39. Процедурные языки ... 1108
39.1. Установка процедурных языков ... 1108
40. PL/pgSQL — процедурный язык SQL ... 1111
40.1. Обзор ... 1111
40.2. Структура PL/pgSQL ... 1112
40.3. Объявления ... 1114
40.4. Выражения ... 1119
40.5. Основные операторы ... 1119
40.6. Управляющие структуры ... 1127
40.7. Курсоры ... 1140
40.8. Сообщения и ошибки ... 1145
40.9. Триггерные процедуры ... 1147
40.10. PL/pgSQL изнутри ... 1155
40.11. Советы по разработке на PL/pgSQL ... 1159
40.12. Портирование из Oracle PL/SQL ... 1161
41. PL/Tcl — процедурный язык Tcl ... 1171
41.1. Обзор ... 1171
41.2. Функции на PL/Tcl и их аргументы ... 1171
41.3. Значения данных в PL/Tcl ... 1173
41.4. Глобальные данные в PL/Tcl ... 1173
41.5. Обращение к базе данных из PL/Tcl ... 1174
41.6. Процедуры триггеров на PL/Tcl ... 1176
41.7. Процедуры событийных триггеров в PL/Tcl ... 1178
41.8. Обработка ошибок в PL/Tcl ... 1178
41.9. Явные подтранзакции в PL/Tcl ... 1179
41.10. Конфигурация PL/Tcl ... 1180
41.11. Имена процедур Tcl ... 1180
42. PL/Perl — процедурный язык Perl ... 1182
42.1. Функции на PL/Perl и их аргументы ... 1182
42.2. Значения в PL/Perl ... 1186
42.3. Встроенные функции ... 1186
42.4. Глобальные значения в PL/Perl ... 1190
42.5. Доверенный и недоверенный PL/Perl ... 1191
42.6. Триггеры на PL/Perl ... 1192
42.7. Событийные триггеры на PL/Perl ... 1194
42.8. Внутренние особенности PL/Perl ... 1194
43. PL/Python — процедурный язык Python ... 1196
43.1. Python 2 и Python 3 ... 1196
43.2. Функции на PL/Python ... 1197

43.3. Значения данных ... 1198
43.4. Совместное использование данных ... 1203
43.5. Анонимные блоки кода ... 1203
43.6. Триггерные функции ... 1203
43.7. Обращение к базе данных ... 1204
43.8. Явные подтранзакции ... 1208
43.9. Вспомогательные функции ... 1209
43.10. Переменные окружения ... 1210
44. Интерфейс программирования сервера ... 1212
44.1. Интерфейсные функции ... 1212
44.2. Вспомогательные интерфейсные функции ... 1245
44.3. Управление памятью ... 1253
44.4. Видимость изменений в данных ... 1263
44.5. Примеры ... 1263
45. Фоновые рабочие процессы ... 1267
46. Логическое декодирование ... 1270
46.1. Примеры логического декодирования ... 1270
46.2. Концепции логического декодирования ... 1272
46.3. Интерфейс протокола потоковой репликации ... 1273
46.4. Интерфейс логического декодирования на уровне SQL ... 1274
46.5. Системные каталоги, связанные с логическим декодированием ... 1274
46.6. Модули вывода логического декодирования ... 1274
46.7. Запись вывода логического декодирования ... 1277
46.8. Поддержка синхронной репликации для логического декодирования ... 1277
47. Отслеживание прогресса репликации ... 1279
VI. Справочное руководство ... 1280
I. Команды SQL ... 1281
ABORT ... 1282
ALTER AGGREGATE ... 1283
ALTER COLLATION ... 1285
ALTER CONVERSION ... 1287
ALTER DATABASE ... 1288
ALTER DEFAULT PRIVILEGES ... 1290
ALTER DOMAIN ... 1293
ALTER EVENT TRIGGER ... 1296
ALTER EXTENSION ... 1297
ALTER FOREIGN DATA WRAPPER ... 1300
ALTER FOREIGN TABLE ... 1302
ALTER FUNCTION ... 1307
ALTER GROUP ... 1310
ALTER INDEX ... 1311
ALTER LANGUAGE ... 1313
ALTER LARGE OBJECT ... 1314
ALTER MATERIALIZED VIEW ... 1315
ALTER OPERATOR ... 1317
ALTER OPERATOR CLASS ... 1319
ALTER OPERATOR FAMILY ... 1320
ALTER POLICY ... 1324
ALTER PUBLICATION ... 1325
ALTER ROLE ... 1327
ALTER RULE ... 1331
ALTER SCHEMA ... 1332
ALTER SEQUENCE ... 1333
ALTER SERVER ... 1336
ALTER STATISTICS ... 1338
ALTER SUBSCRIPTION ... 1339
ALTER SYSTEM ... 1341
ALTER TABLE ... 1343

ALTER TABLESPACE 1357
ALTER TEXT SEARCH CONFIGURATION 1358
ALTER TEXT SEARCH DICTIONARY 1360
ALTER TEXT SEARCH PARSER 1362
ALTER TEXT SEARCH TEMPLATE 1363
ALTER TRIGGER 1364
ALTER TYPE 1365
ALTER USER 1368
ALTER USER MAPPING 1369
ALTER VIEW 1370
ANALYZE 1372
BEGIN 1375
CHECKPOINT 1377
CLOSE 1378
CLUSTER 1379
COMMENT 1381
COMMIT 1385
COMMIT PREPARED 1386
COPY 1387
CREATE ACCESS METHOD 1397
CREATE AGGREGATE 1398
CREATE CAST 1405
CREATE COLLATION 1409
CREATE CONVERSION 1411
CREATE DATABASE 1413
CREATE DOMAIN 1416
CREATE EVENT TRIGGER 1419
CREATE EXTENSION 1421
CREATE FOREIGN DATA WRAPPER 1424
CREATE FOREIGN TABLE 1426
CREATE FUNCTION 1430
CREATE GROUP 1438
CREATE INDEX 1439
CREATE LANGUAGE 1446
CREATE MATERIALIZED VIEW 1449
CREATE OPERATOR 1451
CREATE OPERATOR CLASS 1454
CREATE OPERATOR FAMILY 1457
CREATE POLICY 1458
CREATE PUBLICATION 1464
CREATE ROLE 1466
CREATE RULE 1471
CREATE SCHEMA 1474
CREATE SEQUENCE 1476
CREATE SERVER 1480
CREATE STATISTICS 1482
CREATE SUBSCRIPTION 1484
CREATE TABLE 1487
CREATE TABLE AS 1506
CREATE TABLESPACE 1509
CREATE TEXT SEARCH CONFIGURATION 1511
CREATE TEXT SEARCH DICTIONARY 1512
CREATE TEXT SEARCH PARSER 1514
CREATE TEXT SEARCH TEMPLATE 1516
CREATE TRANSFORM 1517
CREATE TRIGGER 1519
CREATE TYPE 1526
CREATE USER 1535

CREATE USER MAPPING 1536
CREATE VIEW 1537
DEALLOCATE 1542
DECLARE 1543
DELETE 1547
DISCARD 1550
DO 1551
DROP ACCESS METHOD 1552
DROP AGGREGATE 1553
DROP CAST 1555
DROP COLLATION 1556
DROP CONVERSION 1557
DROP DATABASE 1558
DROP DOMAIN 1559
DROP EVENT TRIGGER 1560
DROP EXTENSION 1561
DROP FOREIGN DATA WRAPPER 1562
DROP FOREIGN TABLE 1563
DROP FUNCTION 1564
DROP GROUP 1566
DROP INDEX 1567
DROP LANGUAGE 1569
DROP MATERIALIZED VIEW 1570
DROP OPERATOR 1571
DROP OPERATOR CLASS 1573
DROP OPERATOR FAMILY 1575
DROP OWNED 1576
DROP POLICY 1577
DROP PUBLICATION 1578
DROP ROLE 1579
DROP RULE 1580
DROP SCHEMA 1581
DROP SEQUENCE 1582
DROP SERVER 1583
DROP STATISTICS 1584
DROP SUBSCRIPTION 1585
DROP TABLE 1586
DROP TABLESPACE 1587
DROP TEXT SEARCH CONFIGURATION 1588
DROP TEXT SEARCH DICTIONARY 1589
DROP TEXT SEARCH PARSER 1590
DROP TEXT SEARCH TEMPLATE 1591
DROP TRANSFORM 1592
DROP TRIGGER 1593
DROP TYPE 1594
DROP USER 1595
DROP USER MAPPING 1596
DROP VIEW 1597
END 1598
EXECUTE 1599
EXPLAIN 1600
FETCH 1605
GRANT 1609
IMPORT FOREIGN SCHEMA 1616
INSERT 1618
LISTEN 1625
LOAD 1627
LOCK 1628

MOVE ... 1631
NOTIFY ... 1633
PREPARE ... 1636
PREPARE TRANSACTION ... 1639
REASSIGN OWNED ... 1641
REFRESH MATERIALIZED VIEW ... 1642
REINDEX ... 1644
RELEASE SAVEPOINT ... 1647
RESET ... 1648
REVOKE ... 1649
ROLLBACK ... 1653
ROLLBACK PREPARED ... 1654
ROLLBACK TO SAVEPOINT ... 1655
SAVEPOINT ... 1657
SECURITY LABEL ... 1659
SELECT ... 1661
SELECT INTO ... 1681
SET ... 1683
SET CONSTRAINTS ... 1686
SET ROLE ... 1688
SET SESSION AUTHORIZATION ... 1690
SET TRANSACTION ... 1692
SHOW ... 1695
START TRANSACTION ... 1697
TRUNCATE ... 1698
UNLISTEN ... 1700
UPDATE ... 1701
VACUUM ... 1705
VALUES ... 1708
WAITLSN ... 1711
II. Клиентские приложения Postgres Pro ... 1713
clusterdb ... 1714
createdb ... 1717
createuser ... 1720
dropdb ... 1724
dropuser ... 1726
ecpg ... 1728
pg_basebackup ... 1730
pgbench ... 1737
pg_config ... 1750
pg_dump ... 1753
pg_dumpall ... 1765
pg_isready ... 1771
pg_receivewal ... 1773
pg_recvlogical ... 1777
pg_restore ... 1781
pg-wrapper ... 1790
psql ... 1792
reindexdb ... 1831
vacuumdb ... 1834
III. Серверные приложения Postgres Pro ... 1838
initdb ... 1839
pg_archivecleanup ... 1843
pg_controldata ... 1845
pg_ctl ... 1846
pg_resetwal ... 1852
pg_rewind ... 1855
pg-setup ... 1858

pg_test_fsync 1860
pg_test_timing 1861
pg_upgrade 1865
pg_waldump 1873
postgres 1875
postmaster 1882
VII. Внутреннее устройство 1883
48. Обзор внутреннего устройства Postgres Pro 1884
48.1. Путь запроса 1884
48.2. Как устанавливаются соединения 1884
48.3. Этап разбора 1885
48.4. Система правил Postgres Pro 1886
48.5. Планировщик/оптимизатор 1886
48.6. Исполнитель 1888
49. Системные каталоги 1889
49.1. Обзор 1889
49.2. pg_aggregate 1891
49.3. pg_am 1893
49.4. pg_amop 1894
49.5. pg_amproc 1895
49.6. pg_attrdef 1896
49.7. pg_attribute 1896
49.8. pg_authid 1900
49.9. pg_auth_members 1901
49.10. pg_cast 1901
49.11. pg_class 1903
49.12. pg_collation 1908
49.13. pg_constraint 1909
49.14. pg_conversion 1912
49.15. pg_database 1913
49.16. pg_db_role_setting 1915
49.17. pg_default_acl 1916
49.18. pg_depend 1917
49.19. pg_description 1918
49.20. pg_enum 1919
49.21. pg_event_trigger 1919
49.22. pg_extension 1920
49.23. pg_foreign_data_wrapper 1921
49.24. pg_foreign_server 1922
49.25. pg_foreign_table 1922
49.26. pg_index 1923
49.27. pg_inherits 1926
49.28. pg_init_privs 1926
49.29. pg_language 1927
49.30. pg_largeobject 1929
49.31. pg_largeobject_metadata 1929
49.32. pg_namespace 1930
49.33. pg_opclass 1930
49.34. pg_operator 1931
49.35. pg_opfamily 1932
49.36. pg_partitioned_table 1932
49.37. pg_pltemplate 1934
49.38. pg_policy 1935
49.39. pg_proc 1935
49.40. pg_publication 1941
49.41. pg_publication_rel 1941

49.42. pg_range ... 1941
49.43. pg_replication_origin ... 1942
49.44. pg_rewrite ... 1942
49.45. pg_seclabel ... 1943
49.46. pg_sequence ... 1944
49.47. pg_shdepend ... 1945
49.48. pg_shdescription ... 1946
49.49. pg_shseclabel ... 1946
49.50. pg_statistic ... 1947
49.51. pg_statistic_ext ... 1949
49.52. pg_subscription ... 1950
49.53. pg_subscription_rel ... 1951
49.54. pg_tablespace ... 1951
49.55. pg_transform ... 1952
49.56. pg_trigger ... 1953
49.57. pg_ts_config ... 1955
49.58. pg_ts_config_map ... 1955
49.59. pg_ts_dict ... 1956
49.60. pg_ts_parser ... 1956
49.61. pg_ts_template ... 1957
49.62. pg_type ... 1958
49.63. pg_user_mapping ... 1966
49.64. Системные представления ... 1966
49.65. pg_available_extensions ... 1967
49.66. pg_available_extension_versions ... 1968
49.67. pg_config ... 1968
49.68. pg_cursors ... 1968
49.69. pg_file_settings ... 1969
49.70. pg_group ... 1970
49.71. pg_hba_file_rules ... 1970
49.72. pg_indexes ... 1971
49.73. pg_locks ... 1972
49.74. pg_matviews ... 1976
49.75. pg_policies ... 1976
49.76. pg_prepared_statements ... 1977
49.77. pg_prepared_xacts ... 1978
49.78. pg_publication_tables ... 1979
49.79. pg_replication_origin_status ... 1979
49.80. pg_replication_slots ... 1979
49.81. pg_roles ... 1981
49.82. pg_rules ... 1983
49.83. pg_seclabels ... 1983
49.84. pg_sequences ... 1984
49.85. pg_settings ... 1985
49.86. pg_shadow ... 1987
49.87. pg_stats ... 1988
49.88. pg_tables ... 1991
49.89. pg_timezone_abbrevs ... 1991
49.90. pg_timezone_names ... 1992
49.91. pg_user ... 1992
49.92. pg_user_mappings ... 1993
49.93. pg_views ... 1994
50. Клиент-серверный протокол ... 1995
50.1. Обзор ... 1995
50.2. Поток сообщений ... 1997
50.3. Аутентификация SASL ... 2008

50.4. Протокол потоковой репликации ... 2010
50.5. Протокол логической потоковой репликации ... 2016
50.6. Типы данных в сообщениях ... 2017
50.7. Форматы сообщений ... 2018
50.8. Поля сообщений с ошибками и замечаниями ... 2033
50.9. Форматы сообщений логической репликации ... 2035
50.10. Сводка изменений по сравнению с протоколом версии 2.0 ... 2039
51. Написание обработчика процедурного языка ... 2041
52. Написание обёртки сторонних данных ... 2044
52.1. Функции обёрток сторонних данных ... 2044
52.2. Подпрограммы обёртки сторонних данных ... 2044
52.3. Вспомогательные функции для обёрток сторонних данных ... 2057
52.4. Планирование запросов с обёртками сторонних данных ... 2058
52.5. Блокировка строк в обёртках сторонних данных ... 2060
53. Написание метода извлечения выборки таблицы ... 2063
53.1. Опорные функции метода извлечения выборки ... 2063
54. Написание провайдера нестандартного сканирования ... 2067
54.1. Создание нестандартных путей сканирования ... 2067
54.2. Создание нестандартных планов сканирования ... 2068
54.3. Выполнение нестандартного сканирования ... 2069
55. Генетический оптимизатор запросов ... 2072
55.1. Обработка запроса как сложная задача оптимизации ... 2072
55.2. Генетические алгоритмы ... 2072
55.3. Генетическая оптимизация запросов (GEQO) в Postgres Pro ... 2073
55.4. Дополнительные источники информации ... 2074
56. Определение интерфейса для индексных методов доступа ... 2076
56.1. Базовая структура API для индексов ... 2076
56.2. Функции для индексных методов доступа ... 2078
56.3. Сканирование индекса ... 2084
56.4. Замечания о блокировке с индексами ... 2085
56.5. Проверки уникальности в индексе ... 2087
56.6. Функции оценки стоимости индекса ... 2088
57. Унифицированные записи WAL ... 2091
58. Индексы GiST ... 2093
58.1. Введение ... 2093
58.2. Встроенные классы операторов ... 2093
58.3. Расширяемость ... 2093
58.4. Реализация ... 2102
58.5. Примеры ... 2102
59. Индексы SP-GiST ... 2104
59.1. Введение ... 2104
59.2. Встроенные классы операторов ... 2104
59.3. Расширяемость ... 2104
59.4. Реализация ... 2112
59.5. Примеры ... 2113
60. Индексы GIN ... 2114
60.1. Введение ... 2114
60.2. Встроенные классы операторов ... 2114
60.3. Расширяемость ... 2114
60.4. Реализация ... 2117
60.5. Приёмы и советы по применению GIN ... 2118
60.6. Ограничения ... 2119
60.7. Примеры ... 2119
61. Индексы BRIN ... 2121
61.1. Введение ... 2121
61.2. Встроенные классы операторов ... 2122
61.3. Расширяемость ... 2123
62. Хеш-индексы ... 2126

62.1. Обзор ... 2126
62.2. Реализация ... 2127
63. Физическое хранение базы данных ... 2128
63.1. Размещение файлов базы данных ... 2128
63.2. TOAST ... 2130
63.3. Карта свободного пространства ... 2133
63.4. Карта видимости ... 2134
63.5. Слой инициализации ... 2134
63.6. Компоновка страницы базы данных ... 2134
64. Внутренний интерфейс BKI ... 2138
64.1. Формат файла BKI ... 2138
64.2. Команды BKI ... 2138
64.3. Структура файла BKI ... 2139
64.4. Пример ... 2140
65. Как планировщик использует статистику ... 2141
65.1. Примеры оценки количества строк ... 2141
65.2. Примеры многовариантной статистики ... 2146
65.3. Статистика планировщика и безопасность ... 2147
VIII. Приложения ... 2149
A. Коды ошибок Postgres Pro ... 2150
B. Поддержка даты и времени ... 2159
B.1. Интерпретация данных даты и времени ... 2159
B.2. Обработка недопустимых или неоднозначных значений даты/времени ... 2160
B.3. Ключевые слова для обозначения даты и времени ... 2161
B.4. Файлы конфигурации даты/времени ... 2162
B.5. Указание часовых поясов в стиле POSIX ... 2163
B.6. История единиц измерения времени ... 2165
B.7. Юлианские даты ... 2166
C. Ключевые слова SQL ... 2168
D. Соответствие стандарту SQL ... 2200
D.1. Поддерживаемые возможности ... 2201
D.2. Неподдерживаемые возможности ... 2220
D.3. Ограничения XML и совместимость с SQL/XML ... 2234
E. Замечания к выпускам ... 2238
E.1. Postgres Pro Standard 10.23.1 ... 2238
E.2. Postgres Pro Standard 10.22.1 ... 2239
E.3. Postgres Pro Standard 10.21.1 ... 2240
E.4. Postgres Pro Standard 10.20.1 ... 2242
E.5. Postgres Pro Standard 10.19.1 ... 2243
E.6. Postgres Pro Standard 10.18.1 ... 2244
E.7. Postgres Pro Standard 10.17.1 ... 2246
E.8. Postgres Pro Standard 10.16.1 ... 2247
E.9. Postgres Pro Standard 10.15.1 ... 2248
E.10. Postgres Pro Standard 10.14.1 ... 2249
E.11. Postgres Pro Standard 10.13.1 ... 2251
E.12. Postgres Pro Standard 10.12.1 ... 2252
E.13. Postgres Pro Standard 10.11.1 ... 2253
E.14. Postgres Pro Standard 10.10.2 ... 2254
E.15. Postgres Pro Standard 10.10.1 ... 2255
E.16. Postgres Pro Standard 10.9.1 ... 2255
E.17. Postgres Pro Standard 10.7.1 ... 2257
E.18. Postgres Pro Standard 10.6.1 ... 2258
E.19. Postgres Pro Standard 10.5.3 ... 2260
E.20. Postgres Pro Standard 10.5.2 ... 2261
E.21. Postgres Pro Standard 10.5.1 ... 2261
E.22. Postgres Pro Standard 10.4.1 ... 2262
E.23. Postgres Pro Standard 10.3.2 ... 2264
E.24. Postgres Pro Standard 10.3.1 ... 2264

E.25. Postgres Pro Standard 10.2.1 ... 2265
E.26. Postgres Pro Standard 10.1.1 ... 2265
E.27. Выпуск 10.23 ... 2268
E.28. Выпуск 10.22 ... 2272
E.29. Выпуск 10.21 ... 2275
E.30. Выпуск 10.20 ... 2279
E.31. Выпуск 10.19 ... 2282
E.32. Выпуск 10.18 ... 2288
E.33. Выпуск 10.17 ... 2292
E.34. Выпуск 10.16 ... 2295
E.35. Выпуск 10.15 ... 2299
E.36. Выпуск 10.14 ... 2303
E.37. Выпуск 10.13 ... 2307
E.38. Выпуск 10.12 ... 2311
E.39. Выпуск 10.11 ... 2315
E.40. Выпуск 10.10 ... 2320
E.41. Выпуск 10.9 ... 2323
E.42. Выпуск 10.8 ... 2325
E.43. Выпуск 10.7 ... 2329
E.44. Выпуск 10.6 ... 2334
E.45. Выпуск 10.5 ... 2339
E.46. Выпуск 10.4 ... 2344
E.47. Выпуск 10.3 ... 2349
E.48. Выпуск 10.2 ... 2351
E.49. Выпуск 10.1 ... 2356
E.50. Выпуск 10 ... 2359
E.51. Предыдущие выпуски ... 2381
F. Дополнительно поставляемые модули ... 2382
F.1. adminpack ... 2382
F.2. amcheck ... 2384
F.3. auth_delay ... 2387
F.4. auto_explain ... 2387
F.5. bloom ... 2390
F.6. btree_gin ... 2393
F.7. btree_gist ... 2393
F.8. chkpass ... 2394
F.9. citext ... 2395
F.10. spi ... 2397
F.11. cube ... 2400
F.12. dblink ... 2405
F.13. dict_int ... 2434
F.14. dict_xsyn ... 2434
F.15. dump_stat ... 2436
F.16. earthdistance ... 2438
F.17. fasttrun ... 2440
F.18. file_fdw ... 2440
F.19. fulleq ... 2442
F.20. fuzzystrmatch ... 2443
F.21. hstore ... 2446
F.22. Модули словарей Hunspell ... 2452
F.23. intagg ... 2454
F.24. intarray ... 2455
F.25. isn ... 2458
F.26. jsquery ... 2461
F.27. lo ... 2466
F.28. ltree ... 2467
F.29. mchar ... 2474
F.30. online_analyze ... 2475

F.31. pageinspect ... 2476
F.32. passwordcheck ... 2482
F.33. pg_buffercache ... 2483
F.34. pgcrypto ... 2484
F.35. pg_freespacemap ... 2495
F.36. pg_pathman ... 2496
F.37. pg_prewarm ... 2516
F.38. pg_query_state ... 2517
F.39. pgrowlocks ... 2522
F.40. pg_stat_statements ... 2524
F.41. pgstattuple ... 2529
F.42. pg_trgm ... 2533
F.43. pg_tsparser ... 2538
F.44. pg_variables ... 2539
F.45. pg_visibility ... 2547
F.46. plantuner ... 2548
F.47. postgres_fdw ... 2550
F.48. seg ... 2556
F.49. sepgsql ... 2559
F.50. shared_ispell ... 2566
F.51. snowball_ext ... 2568
F.52. sr_plan ... 2568
F.53. sslinfo ... 2571
F.54. tablefunc ... 2572
F.55. tcn ... 2581
F.56. test_decoding ... 2582
F.57. tsm_system_rows ... 2583
F.58. tsm_system_time ... 2583
F.59. unaccent ... 2584
F.60. uuid-ossp ... 2586
F.61. xml2 ... 2587
G. Дополнительно поставляемые программы ... 2593
G.1. Клиентские приложения ... 2593
G.2. Серверные приложения ... 2661
H. Внешние проекты ... 2720
H.1. Клиентские интерфейсы ... 2720
H.2. Средства администрирования ... 2720
H.3. Процедурные языки ... 2720
H.4. Расширения ... 2721
I. Настройка Postgres Pro для решений 1С ... 2722
J. Демонстрационная база данных «Авиаперевозки» ... 2723
J.1. Установка ... 2723
J.2. Диаграмма схемы данных ... 2724
J.3. Описание схемы ... 2724
J.4. Объекты схемы ... 2725
J.5. Использование ... 2731
K. Сокращения ... 2739
L. Устаревшая или переименованная функциональность ... 2744
L.1. `pg_xlogdump` переименована в `pg_waldump` ... 2744
L.2. `pg_resetxlog` переименована в `pg_resetwal` ... 2744
L.3. `pg_receivexlog` переименована в `pg_receivewal` ... 2744
Библиография ... 2745
Предметный указатель ... 2747

Предисловие

Эта книга является официальной документацией Postgres Pro Standard. Она была написана разработчиками Postgres Pro, сообществом PostgreSQL и другими добровольцами параллельно с разработкой программного продукта PostgreSQL и Postgres Pro. В ней описывается вся функциональность, которую официально поддерживает текущая версия Postgres Pro.

Чтобы такой большой объём информации о Postgres Pro был удобоварим, эта книга разделена на части. Каждая часть предназначена определённой категории пользователей или пользователям на разных стадиях освоения Postgres Pro:

- Часть I представляет собой неформальное введение для начинающих.
- Часть II описывает язык SQL и его среду, включая типы данных и функции, а также оптимизацию производительности на уровне пользователей. Это будет полезно прочитать всем пользователям Postgres Pro.
- Часть III описывает установку и администрирование сервера. Эта часть будет полезна всем, кто использует сервер Postgres Pro для самых разных целей.
- Часть IV описывает программные интерфейсы для клиентских приложений Postgres Pro.
- Часть V предназначена для более опытных пользователей и содержит сведения о возможностях расширения сервера. Эта часть включает темы, посвящённые, в частности, пользовательским типам данных и функциям.
- Часть VI содержит справочную информацию о командах SQL, а также клиентских и серверных программах. Эта часть дополняет другие информацией, структурированной по командам и программам.
- Часть VII содержит разнообразную информацию, полезную для разработчиков Postgres Pro.

1. Что такое Postgres Pro Standard?

Postgres Pro Standard — это объектно-реляционная система управления базами данных (ОРСУБД, ORDBMS), разработанная Postgres Professional в рамках проекта Postgres Pro на основе *PostgreSQL*, в свою очередь, основанном на *POSTGRES, Version 4.2* — программе, разработанной на факультете компьютерных наук Калифорнийского университета в Беркли. В POSTGRES появилось множество новшеств, которые были реализованы в некоторых коммерческих СУБД гораздо позднее.

Postgres Pro Standard, как и PostgreSQL поддерживает большую часть стандарта SQL и предлагает множество современных функций:

- сложные запросы
- внешние ключи
- триггеры
- изменяемые представления
- транзакционная целостность
- многоверсионность

Кроме того, пользователи могут всячески расширять Postgres Pro, так же как и PostgreSQL, например, создавая свои

- типы данных
- функции
- операторы
- агрегатные функции
- методы индексирования
- процедурные языки

2. Различия между Postgres Pro Standard и PostgreSQL

Postgres Pro предоставляет наиболее актуальную версию PostgreSQL с дополнительными изменениями и расширениями. Этот продукт включает все новые возможности, реализованные

компанией Postgres Professional, а также сторонние доработки, которые уже приняты сообществом PostgreSQL и попадут в новые версии PostgreSQL. Таким образом, пользователи Postgres Pro Standard получают ранний доступ к важным нововведениям и исправлениям.

> **Примечание**
>
> Postgres Pro Standard предоставляется по следующей лицензии: *https://postgrespro.ru/products/postgrespro/eula*. Обязательно ознакомьтесь с условиями лицензии, прежде чем загружать и использовать Postgres Pro Standard.

Postgres Pro Standard отличают от PostgreSQL следующие усовершенствования:

- Покрывающие индексы. (См. раздел, посвящённый `INCLUDE`, в описании CREATE INDEX.)
- Улучшенный механизм проверки блокировок, не оказывающий отрицательного влияния на производительность.
- Уменьшенное потребление памяти при обработке сложных запросов со множеством таблиц.
- Добавление времени планирования в информацию, выводимую модулем auto_explain.
- Возможность замены нулевого байта заданным ASCII-символом при загрузке данных с помощью команды `COPY FROM`. (См. описание параметра nul_byte_replacement_on_import.)
- Использование ICU на всех платформах с целью обеспечить платформонезависимую сортировку для различных локалей. По умолчанию провайдер правил сортировки `icu` задействуется для всех локалей, за исключением `C` и `POSIX`. (См. Подраздел 22.2.2.)
- Реализация механизма PTRACK, позволяющего программе pg_probackup отслеживать изменения страниц при создании инкрементальных резервных копий.
- Изменение параметров в `recovery.conf` без перезапуска сервера.
- Согласованное чтение на ведомых серверах. (См. WAITLSN.)
- Улучшения редактирования в командной строке с использованием WinEditLine в версии psql для Windows, в том числе поддержка автодополнение в консоли psql и изменение кодировки psql по умолчанию на UTF-8.
- Унифицированная структура пакетов двоичных файлов для всех дистрибутивов Linux, упрощающая миграцию между ними и позволяющая устанавливать несколько различных продуктов на базе PostgreSQL совместно без каких-либо конфликтов. (См. Главу 16.)

Postgres Pro Standard также включает следующие дополнительные модули:

- Модуль dump_stat, позволяющий сохранять статистику данных при резервном копировании и восстановлении.
- Модуль fasttrun, который предоставляет транзакционно-небезопасную функцию для усечения временных таблиц, что предотвращает разрастание каталога `pg_class`.
- Модуль fulleq, предоставляющий дополнительный оператор равенства для совместимости с Microsoft SQL Server.
- Модуль hunspell-dict, предоставляющий словари для ряда языков.
- Модуль jsquery реализует специальный язык запросов для эффективного, с использованием индексов, поиска в структурированных данных JSONB
- Служба мониторинга mamonsu, исполненная в виде агента Zabbix.
- Модуль mchar, предоставляющий дополнительный тип данных для совместимости с Microsoft SQL Server.
- Модуль online_analyze, привносящий набор функций, которые немедленно обновляют статистику в целевых таблицах после операций `INSERT`, `UPDATE`, `DELETE` или `SELECT INTO` в них.
- Пул соединений pgbouncer.
- Модуль pg_pathman предоставляет оптимизированный механизм секционирования, а также функции для создания и управления секциями.
- pg_probackup — менеджер резервного копирования и восстановления.
- Утилита pgpro_controldata, показывающая управляющую информацию кластера БД PostgreSQL/Postgres Pro и параметры совместимости кластера и/или сервера.
- Модуль pg_query_state, дающий возможность узнавать текущее состояние выполнения запросов в обслуживающем процессе.

- Утилита pg_repack для реорганизации таблиц.
- Модуль pg_variables, предоставляющий функции для работы с переменными различных типов.
- Модуль pg_tsparser — альтернативный анализатор текстового поиска.
- Модуль plantuner, добавляющий поддержку указаний для планировщика, подключающих или отключающих определённые индексы при выполнении запроса.
- Модуль shared_ispell, позволяющий разместить словари в общей памяти.
- Модуль sr_plan, позволяющий сохранять и восстанавливать планы запросов.

Выпуски Postgres Pro Standard следуют за выпусками PostgreSQL, хотя иногда могут выпускаться чаще. Схема версионирования Postgres Pro Standard основана на схеме версионирования PostgreSQL и включает дополнительную цифру.

3. Краткая история PostgreSQL

Объектно-реляционная система управления базами данных, именуемая сегодня PostgreSQL, произошла от пакета POSTGRES, написанного в Беркли, Калифорнийском университете. После двух десятилетий разработки PostgreSQL стал самой развитой СУБД с открытым исходным кодом.

3.1. Проект POSTGRES в Беркли

Проект POSTGRES, возглавляемый профессором Майклом Стоунбрейкером, спонсировали агентство DARPA при Минобороны США, Управление военных исследований (ARO), Национальный Научный Фонд (NSF) и компания ESL, Inc. Реализация POSTGRES началась в 1986 г. Первоначальные концепции системы были представлены в документе ston86, а описание первой модели данных появилось в rowe87. Проект системы правил тогда был представлен в ston87a. Суть и архитектура менеджера хранилища были расписаны в ston87b.

С тех пор POSTGRES прошёл несколько этапов развития. Первая «демоверсия» заработала в 1987 и была показана в 1988 на конференции ACM-SIGMOD. Версия 1, описанная в ston90a, была выпущена для нескольких внешних пользователей в июне 1989. В ответ на критику первой системы правил (ston89), она была переделана (ston90b), и в версии 2, выпущенной в июне 1990, была уже новая система правил. В 1991 вышла версия 3, в которой появилась поддержка различных менеджеров хранилища, улучшенный исполнитель запросов и переписанная система правил. Последующие выпуски до Postgres95 (см. ниже) в основном были направлены на улучшение портируемости и надёжности.

POSTGRES применялся для реализации множества исследовательских и производственных задач. В их числе: система анализа финансовых данных, пакет мониторинга работы реактивных двигателей, база данных наблюдений за астероидами, база данных медицинской информации, несколько географических информационных систем. POSTGRES также использовался для обучения в нескольких университетах. Наконец, компания Illustra Information Technologies (позже ставшая частью *Informix*, которая сейчас принадлежит *IBM*) воспользовалась кодом и нашла ему коммерческое применение. В конце 1992 POSTGRES стал основной СУБД научного вычислительного проекта *Sequoia 2000*.

В 1993 число внешних пользователей удвоилось. Стало очевидно, что обслуживание кода и поддержка занимает слишком много времени, и его не хватает на исследования. Для снижения этой нагрузки проект POSTGRES в Беркли был официально закрыт на версии 4.2.

3.2. Postgres95

В 1994 Эндри Ю и Джолли Чен добавили в POSTGRES интерпретатор языка SQL. Уже с новым именем Postgres95 был опубликован в Интернете и начал свой путь как потомок разработанного в Беркли POSTGRES, с открытым исходным кодом.

Код Postgres95 был приведён в полное соответствие с ANSI C и уменьшился на 25%. Благодаря множеству внутренних изменений он стал быстрее и удобнее. Postgres95 версии 1.0.x работал примерно на 30-50% быстрее POSTGRES версии 4.2 (по тестам Wisconsin Benchmark). Помимо исправления ошибок, произошли следующие изменения:

- На смену языку запросов PostQUEL пришёл SQL (реализованный в сервере). (Интерфейсная библиотека libpq унаследовала своё имя от PostQUEL.) Подзапросы не поддерживались до выхода PostgreSQL (см. ниже), хотя их можно было имитировать в Postgres95 с помощью пользовательских функций SQL. Были заново реализованы агрегатные функции. Также появилась поддержка предложения `GROUP BY`.
- Для интерактивных SQL-запросов была разработана новая программа (psql), которая использовала GNU Readline. Старая программа monitor стала не нужна.
- Появилась новая клиентская библиотека `libpgtcl` для поддержки Tcl-клиентов. Пример оболочки, `pgtclsh`, представлял новые команды Tcl для взаимодействия программ Tcl с сервером Postgres95.
- Был усовершенствован интерфейс для работы с большими объектами. Единственным механизмом хранения таких данных стали инверсионные объекты. (Инверсионная файловая система была удалена.)
- Удалена система правил на уровне экземпляров; перезаписывающие правила сохранились.
- С исходным кодом стали распространяться краткие описания возможностей стандартного SQL, а также самого Postgres95.
- Для сборки использовался GNU make (вместо BSD make). Кроме того, стало возможно скомпилировать Postgres95 с немодифицированной версией GCC (было исправлено выравнивание данных).

3.3. PostgreSQL

В 1996 г. стало понятно, что имя «Postgres95» не выдержит испытание временем. Мы выбрали новое имя, PostgreSQL, отражающее связь между оригинальным POSTGRES и более поздними версиями с поддержкой SQL. В то же время, мы продолжили нумерацию версий с 6.0, вернувшись к последовательности, начатой в проекте Беркли POSTGRES.

Многие продолжают называть PostgreSQL именем «Postgres» (теперь уже редко заглавными буквами) по традиции или для простоты. Это название закрепилось как псевдоним или неформальное обозначение.

В процессе разработки Postgres95 основными задачами были поиск и понимание существующих проблем в серверном коде. С переходом к PostgreSQL акценты сместились к реализации новых функций и возможностей, хотя работа продолжается во всех направлениях.

4. Соглашения

Следующие соглашения используются в описаниях команд: квадратные скобки (`[` и `]`) обозначают необязательные части. Фигурные скобки (`{` и `}`) и вертикальная черта (`|`) обозначают выбор одного из предложенных вариантов. Многоточие (`...`) означает, что предыдущий элемент можно повторить.

Иногда для ясности команды SQL предваряются приглашением `=>`, а команды оболочки — приглашением `$`.

Под *администратором* здесь обычно понимается человек, ответственный за установку и запуск сервера, тогда как *пользователь* — кто угодно, кто использует или желает использовать любой компонент системы Postgres Pro. Эти роли не следует воспринимать слишком узко, так как в этой книге нет определённых допущений относительно процедур системного администрирования.

5. Как правильно сообщить об ошибке

Если вы найдёте ошибку в Postgres Pro, дайте нам знать о ней. Благодаря вашему отчёту об ошибке, Postgres Pro станет ещё более надёжным, ведь даже при самом высоком качестве кода нельзя гарантировать, что каждый блок и каждая функция Postgres Pro будет работать везде и при любых обстоятельствах.

Следующие предложения призваны помочь вам в составлении отчёта об ошибке, который можно будет обработать эффективно. Мы не требуем их неукоснительного выполнения, но всё же лучше следовать им для общего блага.

5.1. Диагностика ошибок

Прежде чем сообщать об ошибке, пожалуйста, прочитайте и перечитайте документацию и убедитесь, что вообще возможно сделать то, что вы хотите. Если из документации неясно, можно это сделать или нет, пожалуйста, сообщите и об этом (тогда это ошибка в документации). Если выясняется, что программа делает что-то не так, как написано в документации, это тоже ошибка. Вот лишь некоторые примеры возможных ошибок:

- Программа завершается с аварийным сигналом или сообщением об ошибке операционной системы, указывающей на проблему в программе. (В качестве контрпримера можно привести сообщение «Нет места на диске» — эту проблему вы должны решить сами.)
- Программа выдаёт неправильный результат для любых вводимых данных.
- Программа отказывается принимать допустимые (согласно документации) данные.
- Программа принимает недопустимые данные без сообщения об ошибке или предупреждения. Но помните: то, что вы считаете недопустимым, мы можем считать приемлемым расширением или совместимым с принятой практикой.
- Не удаётся установить Postgres Pro на поддерживаемой платформе, выполняя соответствующие инструкции.

Здесь под «программой» подразумевается произвольный исполняемый файл, а не исключительно серверный процесс.

Медленная работа или высокая загрузка ресурсов — это не обязательно ошибка. Также может не быть ошибкой какое-то несоответствие стандарту SQL, если только явно не декларируется соответствие в данном аспекте.

5.2. Что сообщать

Сообщая о проблеме, постарайтесь изложить все факты. В каждом отчёте об ошибке следует указать:

- Точную последовательность действий для воспроизведения проблемы, начиная с *запуска программы*. Она должна быть самодостаточной; если вывод зависит от данных в таблицах, то недостаточно сообщить один лишь `SELECT`, без предшествующих операторов `CREATE TABLE` и `INSERT`.

  Лучший формат теста для проблем с SQL — файл, который можно передать программе psql и увидеть проблему. (И убедитесь, что в вашем файле `~/.psqlrc` ничего нет.) Самый простой способ получить такой файл — выгрузить объявления таблиц и данные, необходимые для создания полигона, с помощью pg_dump, а затем добавить проблемный запрос. Постарайтесь сократить размер вашего тестового примера, хотя это не абсолютно необходимо. Если ошибка воспроизводится, мы найдём её в любом случае.

  Если ваше приложение использует какой-то другой клиентский интерфейс, например PHP, пожалуйста, попытайтесь свести ошибку к проблемным запросам.

- Результат, который вы получаете. Если есть сообщение об ошибке, покажите его. Если программа завершается ошибкой операционной системы, сообщите о ней. Или если ничего не происходит, отразите это. Даже если в результате вашего теста происходит сбой программы или что-то очевидное, мы можем не наблюдать этого у себя. Проще всего будет скопировать текст с терминала, если это возможно.

> **Примечание**
>
> Если вы упоминаете сообщение об ошибке, пожалуйста, укажите его в наиболее полной форме. Например, в psql, для этого сначала выполните `\set VERBOSITY verbose`.

Если вы цитируете сообщения из журнала событий сервера, присвойте параметру выполнения log_error_verbosity значение `verbose`, чтобы журнал был наиболее подробным.

Примечание

В случае фатальных ошибок сообщение на стороне клиента может не содержать всю необходимую информацию. Пожалуйста, также изучите журнал сервера баз данных.

- Очень важно отметить, какой результат вы ожидали получить. Пожалуйста, приводите ожидаемый результат, если это уместно.
- Все параметры командной строки и другие параметры запуска, включая все связанные переменные окружения или файлы конфигурации, которые вы изменяли.
- Всё, что вы делали не так, как написано в инструкциях по установке.
- Версию Postgres Pro. Чтобы выяснить версию сервера, к которому вы подключены, можно выполнить команду `SELECT pgpro_version();`. Большинство исполняемых программ также поддерживают параметр `--version`; как минимум должно работать `postgres --version` и `psql --version`.
- Сведения о платформе, включая название и версию ядра, библиотеки C, характеристики процессора, памяти и т. д.

5.3. Куда сообщать

В общем случае посылать сообщения об ошибках следует по адресу `<bugs@postgrespro.ru>`. Мы просим вас написать информативную тему письма, возможно включив в неё часть сообщения об ошибке.

Не отправляйте отчёты об ошибках, специфичных для Postgres Pro, на адрес поддержки PostgreSQL, так как Postgres Pro не поддерживается сообществом PostgreSQL. Однако вы можете сообщать по адресу `<pgsql-bugs@lists.postgresql.org>` обо всех ошибках, связанных с PostgreSQL.

Даже если ваша ошибка не связана с особенностями Postgres Pro, не посылайте сообщения в списки рассылки для пользователей, например в `<pgsql-sql@lists.postgresql.org>` или `<pgsql-general@lists.postgresql.org>`. Эти рассылки предназначены для ответов на вопросы пользователей, и их подписчики обычно не хотят получать сообщения об ошибках, более того, они вряд ли исправят их.

Также, пожалуйста, *не* отправляйте отчёты об ошибках в список `<pgsql-hackers@lists.postgresql.org>`. Этот список предназначен для обсуждения разработки PostgreSQL, и будет лучше, если сообщения об ошибках будут существовать отдельно. Мы перенесём обсуждение вашей ошибки в `pgsql-hackers`, если связанная с PostgreSQL проблема потребует дополнительного рассмотрения.

Часть I. Введение

Добро пожаловать на встречу с Postgres Pro. В следующих главах вы сможете получить общее представление о Postgres Pro, реляционных базах данных и языке SQL, если вы ещё не знакомы с этими темами. Этот материал будет понятен даже тем, кто обладает только общими знаниями о компьютерах. Ни опыт программирования, ни навыки использования Unix-систем не требуются. Основная цель этой части — познакомить вас на практике с ключевыми аспектами системы Postgres Pro, поэтому затрагиваемые в ней темы не будут рассматриваться максимально глубоко и полно.

Прочитав это введение, вы, возможно, захотите перейти к Части II, чтобы получить более формализованное знание языка SQL, или к Части IV, посвящённой разработке приложений для Postgres Pro. Для изучения SQL можно воспользоваться демонстрационной базой данных, описанной в Приложении J. Тем же, кто устанавливает и администрирует сервер самостоятельно, следует также прочитать Часть III.

Глава 1. Начало

1.1. Установка

Прежде чем вы сможете использовать Postgres Pro, вы конечно должны его установить. Однако возможно, что Postgres Pro уже установлен у вас, либо потому что он включён в вашу операционную систему, либо его установил системный администратор. Если это так, обратитесь к документации по операционной системе или к вашему администратору и узнайте, как получить доступ к Postgres Pro.

Если вы устанавливаете Postgres Pro Standard самостоятельно, воспользуйтесь инструкциями по установке (Глава 16), а закончив установку, вернитесь к этому введению. Обязательно прочитайте и выполните указания по установке соответствующих переменных окружения.

Если ваш администратор выполнил установку не с параметрами по умолчанию, вам может потребоваться проделать дополнительную работу. Например, если сервер баз данных установлен на удалённом компьютере, вам нужно будет указать в переменной окружения `PGHOST` имя этого компьютера. Вероятно, также придётся установить переменную окружения `PGPORT`. То есть, если вы пытаетесь запустить клиентское приложение и оно сообщает, что не может подключиться к базе данных, вы должны обратиться к вашему администратору. Если это вы сами, вам следует обратиться к документации и убедиться в правильности настройки окружения. Если вы не поняли, о чём здесь идёт речь, перейдите к следующему разделу.

1.2. Основы архитектуры

Прежде чем продолжить, вы должны разобраться в основах архитектуры системы Postgres Pro. Составив картину взаимодействия частей Postgres Pro, вы сможете лучше понять материал этой главы.

Говоря техническим языком, Postgres Pro реализован в архитектуре клиент-сервер. Рабочий сеанс Postgres Pro включает следующие взаимодействующие процессы (программы):

- Главный серверный процесс, управляющий файлами баз данных, принимающий подключения клиентских приложений и выполняющий различные запросы клиентов к базам данных. Эта программа сервера БД называется `postgres`.
- Клиентское приложение пользователя, желающее выполнять операции в базе данных. Клиентские приложения могут быть очень разнообразными: это может быть текстовая утилита, графическое приложение, веб-сервер, использующий базу данных для отображения веб-страниц, или специализированный инструмент для обслуживания БД. Некоторые клиентские приложения поставляются в составе дистрибутива Postgres Pro, однако большинство создают сторонние разработчики.

Как и в других типичных клиент-серверных приложениях, клиент и сервер могут располагаться на разных компьютерах. В этом случае они взаимодействуют по сети TCP/IP. Важно не забывать это и понимать, что файлы, доступные на клиентском компьютере, могут быть недоступны (или доступны только под другим именем) на компьютере-сервере.

Сервер Postgres Pro может обслуживать одновременно несколько подключений клиентов. Для этого он запускает («порождает») отдельный процесс для каждого подключения. Можно сказать, что клиент и серверный процесс общаются, не затрагивая главный процесс `postgres`. Таким образом, главный серверный процесс всегда работает и ожидает подключения клиентов, принимая которые, он организует взаимодействие клиента и отдельного серверного процесса. (Конечно всё это происходит незаметно для пользователя, а эта схема рассматривается здесь только для понимания.)

1.3. Создание базы данных

Первое, как можно проверить, есть ли у вас доступ к серверу баз данных, — это попытаться создать базу данных. Работающий сервер Postgres Pro может управлять множеством баз данных, что позволяет создавать отдельные базы данных для разных проектов и пользователей.

Возможно, ваш администратор уже создал базу данных для вас. В этом случае вы можете пропустить этот этап и перейти к следующему разделу.

Для создания базы данных, в этом примере названной `mydb`, выполните следующую команду:

```
$ createdb mydb
```

Если вы не увидите никаких сообщений, значит операция была выполнена успешно и продолжение этого раздела можно пропустить.

Если вы видите сообщение типа:

```
createdb: command not found
```

значит Postgres Pro не был установлен правильно. Либо он не установлен вообще, либо в путь поиска команд оболочки не включён его каталог. Попробуйте вызвать ту же команду, указав абсолютный путь:

```
$ /usr/local/pgsql/bin/createdb mydb
```

У вас этот путь может быть другим. Свяжитесь с вашим администратором или проверьте, как были выполнены инструкции по установке, чтобы исправить ситуацию.

Ещё один возможный ответ:

```
createdb: не удалось подключиться к базе postgres:
  не удалось подключиться к серверу: No such file or directory
        Он действительно работает локально и принимает
        соединения через Unix-сокет "/tmp/.s.PGSQL.5432"?
```

Это означает, что сервер не работает или `createdb` не может к нему подключиться. И в этом случае пересмотрите инструкции по установке или обратитесь к администратору.

Также вы можете получить сообщение:

```
createdb: не удалось подключиться к базе postgres:
  ВАЖНО:  роль "joe" не существует
```

где фигурирует ваше имя пользователя. Это говорит о том, что администратор не создал учётную запись Postgres Pro для вас. (Учётные записи Postgres Pro отличаются от учётных записей пользователей операционной системы.) Если вы сами являетесь администратором, прочитайте Главу 20, где написано, как создавать учётные записи. Для создания нового пользователя вы должны стать пользователем операционной системы, под именем которого был установлен Postgres Pro (обычно это `postgres`). Также возможно, что вам назначено имя пользователя Postgres Pro, не совпадающее с вашим именем в ОС; в этом случае вам нужно явно указать ваше имя пользователя Postgres Pro, используя ключ `-U` или установив переменную окружения `PGUSER`.

Если у вас есть учётная запись пользователя, но нет прав на создание базы данных, вы увидите сообщение:

```
createdb: создать базу данных не удалось:
  ОШИБКА:  нет прав на создание базы данных
```

Создавать базы данных разрешено не всем пользователям. Если Postgres Pro отказывается создавать базы данных для вас, значит вам необходимо соответствующее разрешение. В этом случае обратитесь к вашему администратору. Если вы устанавливали Postgres Pro сами, то для целей этого введения вы должны войти в систему с именем пользователя, запускающего сервер БД. [1]

[1] Объяснить это поведение можно так: Учётные записи пользователей Postgres Pro отличаются от учётных записей операционной системы. При подключении к базе данных вы можете указать, с каким именем пользователя Postgres Pro нужно подключаться. По умолчанию же используется имя, с которым вы зарегистрированы в операционной системе. При этом получается, что в Postgres Pro всегда есть учётная запись с именем, совпадающим с именем системного

Вы также можете создавать базы данных с другими именами. Postgres Pro позволяет создавать сколько угодно баз данных. Имена баз данных должны начинаться с буквы и быть не длиннее 63 символов. В качестве имени базы данных удобно использовать ваше текущее имя пользователя. Многие утилиты предполагают такое имя по умолчанию, так что вы сможете упростить ввод команд. Чтобы создать базу данных с таким именем, просто введите:

```
$ createdb
```

Если вы больше не хотите использовать вашу базу данных, вы можете удалить её. Например, если вы владелец (создатель) базы данных `mydb`, вы можете уничтожить её, выполнив следующую команду:

```
$ dropdb mydb
```

(Эта команда не считает именем БД по умолчанию имя текущего пользователя, вы должны явно указать его.) В результате будут физически удалены все файлы, связанные с базой данных, и так как отменить это действие нельзя, не выполняйте его, не подумав о последствиях.

Узнать о командах `createdb` и `dropdb` больше можно в справке createdb и dropdb.

1.4. Подключение к базе данных

Создав базу данных, вы можете обратиться к ней:

- Запустив терминальную программу Postgres Pro под названием *psql*, в которой можно интерактивно вводить, редактировать и выполнять команды SQL.
- Используя существующие графические инструменты, например, pgAdmin или офисный пакет с поддержкой ODBC или JDBC, позволяющий создавать и управлять базой данных. Эти возможности здесь не рассматриваются.
- Написав собственное приложение, используя один из множества доступных языковых интерфейсов. Подробнее это рассматривается в Части IV.

Чтобы работать с примерами этого введения, начните с `psql`. Подключиться с его помощью к базе данных `mydb` можно, введя команду:

```
$ psql mydb
```

Если имя базы данных не указать, она будет выбрана по имени пользователя. Об этом уже рассказывалось в предыдущем разделе, посвящённом команде `createdb`.

В `psql` вы увидите следующее сообщение:

```
psql (10.23.1)
Type "help" for help.

mydb=>
```

Последняя строка может выглядеть и так:

```
mydb=#
```

Что показывает, что вы являетесь суперпользователем, и так скорее всего будет, если вы устанавливали экземпляр Postgres Pro сами. В этом случае на вас не будут распространяться никакие ограничения доступа, но для целей данного введения это не важно.

Если вы столкнулись с проблемами при запуске `psql`, вернитесь к предыдущему разделу. Команды `createdb` и `psql` подключаются к серверу одинаково, так что если первая работает, должна работать и вторая.

Последняя строка в выводе `psql` — это приглашение, которое показывает, что `psql` ждёт ваших команд и вы можете вводить SQL-запросы в рабочей среде `psql`. Попробуйте эти команды:

```
mydb=> SELECT pgpro_version();
```

пользователя, запускающего сервер, и к тому же этот пользователь всегда имеет права на создание баз данных. И чтобы подключиться с именем этого пользователя Postgres Pro, необязательно входить с этим именем в систему, достаточно везде передавать его с параметром `-U`.

```
                                                version
-------------------------------------------------------------------------------------------
 PostgresPro 10.23.1 on x86_64-pc-linux-gnu, compiled by gcc (Debian 4.9.2-10) 4.9.2,
 64-bit
(1 row)

mydb=> SELECT current_date;
    date
------------
 2016-01-07
(1 row)

mydb=> SELECT 2 + 2;
 ?column?
----------
        4
(1 row)
```

В программе `psql` есть множество внутренних команд, которые не являются SQL-операторами. Они начинаются с обратной косой черты, «\». Например, вы можете получить справку по различным SQL-командам Postgres Pro, введя:

```
mydb=> \h
```

Чтобы выйти из `psql`, введите:

```
mydb=> \q
```

и `psql` завершит свою работу, а вы вернётесь в командную оболочку операционной системы. (Чтобы узнать о внутренних командах, введите `\?` в приглашении командной строки `psql`.) Все возможности `psql` документированы в справке psql. В этом руководстве мы не будем использовать эти возможности явно, но вы можете изучить их и применять при удобном случае.

Глава 2. Язык SQL

2.1. Введение

В этой главе рассматривается использование SQL для выполнения простых операций. Она призвана только познакомить вас с SQL, но ни в коей мере не претендует на исчерпывающее руководство. Про SQL написано множество книг, включая melt93 и date97. При этом следует учитывать, что некоторые возможности языка Postgres Pro являются расширениями стандарта.

В следующих примерах мы предполагаем, что вы создали базу данных `mydb`, как описано в предыдущей главе, и смогли запустить psql.

Примеры этого руководства также можно найти в пакете исходного кода Postgres Pro в каталоге `src/tutorial/`. (В двоичных дистрибутивах Postgres Pro эти файлы могут не поставляться.) Чтобы использовать эти файлы, перейдите в этот каталог и запустите make:

```
$ cd ..../src/tutorial
$ make
```

При этом будут созданы скрипты и скомпилированы модули C, содержащие пользовательские функции и типы. Затем, чтобы начать работу с учебным материалом, выполните следующее:

```
$ cd ..../tutorial
$ psql -s mydb

...


mydb=> \i basics.sql
```

Команда `\i` считывает и выполняет команды из заданного файла. Переданный `psql` параметр `-s` переводит его в пошаговый режим, когда он делает паузу перед отправкой каждого оператора серверу. Команды, используемые в этом разделе, содержатся в файле `basics.sql`.

2.2. Основные понятия

Postgres Pro — это *реляционная система управления базами данных* (РСУБД). Это означает, что это система управления данными, представленными в виде *отношений* (relation). Отношение — это математически точное обозначение *таблицы*. Хранение данных в таблицах так распространено сегодня, что это кажется самым очевидным вариантом, хотя есть множество других способов организации баз данных. Например, файлы и каталоги в Unix-подобных операционных системах образуют иерархическую базу данных, а сегодня активно развиваются объектно-ориентированные базы данных.

Любая таблица представляет собой именованный набор *строк*. Все строки таблицы имеют одинаковый набор именованных *столбцов*, при этом каждому столбцу назначается определённый тип данных. Хотя порядок столбцов во всех строках фиксирован, важно помнить, что SQL не гарантирует какой-либо порядок строк в таблице (хотя их можно явно отсортировать при выводе).

Таблицы объединяются в базы данных, а набор баз данных, управляемый одним экземпляром сервера Postgres Pro, образует *кластер* баз данных.

2.3. Создание таблицы

Вы можете создать таблицу, указав её имя и перечислив все имена столбцов и их типы:

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- минимальная температура дня
    temp_hi         int,           -- максимальная температура дня
```

```
    prcp            real,          -- уровень осадков
    date            date
);
```

Весь этот текст можно ввести в `psql` вместе с символами перевода строк. `psql` понимает, что команда продолжается до точки с запятой.

В командах SQL можно свободно использовать пробельные символы (пробелы, табуляции и переводы строк). Это значит, что вы можете ввести команду, выровняв её по-другому или даже уместив в одной строке. Два минуса («--») обозначают начало комментария. Всё, что идёт за ними до конца строки, игнорируется. SQL не чувствителен к регистру в ключевых словах и идентификаторах, за исключением идентификаторов, взятых в кавычки (в данном случае это не так).

`varchar(80)` определяет тип данных, допускающий хранение произвольных символьных строк длиной до 80 символов. `int` — обычный целочисленный тип. `real` — тип для хранения чисел с плавающей точкой одинарной точности. `date` — тип даты. (Да, столбец типа `date` также называется `date`. Это может быть удобно или вводить в заблуждение — как посмотреть.)

Postgres Pro поддерживает стандартные типы SQL: `int`, `smallint`, `real`, `double precision`, `char(`*N*`)`, `varchar(`*N*`)`, `date`, `time`, `timestamp` и `interval`, а также другие универсальные типы и богатый набор геометрических типов. Кроме того, Postgres Pro можно расширять, создавая набор собственных типов данных. Как следствие, имена типов не являются ключевыми словами в данной записи, кроме тех случаев, когда это требуется для реализации особых конструкций стандарта SQL.

Во втором примере мы сохраним в таблице города и их географическое положение:

```
CREATE TABLE cities (
    name            varchar(80),
    location        point
);
```

Здесь `point` — пример специфического типа данных Postgres Pro.

Наконец, следует сказать, что если вам больше не нужна какая-либо таблица, или вы хотите пересоздать её по-другому, вы можете удалить её, используя следующую команду:

```
DROP TABLE имя_таблицы;
```

2.4. Добавление строк в таблицу

Для добавления строк в таблицу используется оператор `INSERT`:

```
INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

Заметьте, что для всех типов данных применяются довольно очевидные форматы. Константы, за исключением простых числовых значений, обычно заключаются в апострофы (`'`), как в данном примере. Тип `date` на самом деле очень гибок и принимает разные форматы, но в данном введении мы будем придерживаться простого и однозначного.

Тип `point` требует ввода пары координат, например таким образом:

```
INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
```

Показанный здесь синтаксис требует, чтобы вы запомнили порядок столбцов. Можно также применить альтернативную запись, перечислив столбцы явно:

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
    VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

Вы можете перечислить столбцы в другом порядке, если желаете опустить некоторые из них, например, если уровень осадков (столбец prcp) неизвестен:

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
    VALUES ('1994-11-29', 'Hayward', 54, 37);
```

Многие разработчики предпочитают явно перечислять столбцы, а не полагаться на их порядок в таблице.

Пожалуйста, введите все показанные выше команды, чтобы у вас были данные, с которыми можно будет работать дальше.

Вы также можете загрузить большой объём данных из обычных текстовых файлов, применив команду `COPY`. Обычно это будет быстрее, так как команда `COPY` оптимизирована для такого применения, хотя и менее гибка, чем `INSERT`. Например, её можно использовать так:

```
COPY weather FROM '/home/user/weather.txt';
```

здесь подразумевается, что данный файл доступен на компьютере, где работает серверный процесс, а не на клиенте, так как указанный файл будет прочитан непосредственно на сервере. Подробнее об этом вы можете узнать в описании команды COPY.

2.5. Выполнение запроса

Чтобы получить данные из таблицы, нужно выполнить *запрос*. Для этого предназначен SQL-оператор `SELECT`. Он состоит из нескольких частей: выборки (в которой перечисляются столбцы, которые должны быть получены), списка таблиц (в нём перечисляются таблицы, из которых будут получены данные) и необязательного условия (определяющего ограничения). Например, чтобы получить все строки таблицы `weather`, введите:

```
SELECT * FROM weather;
```

Здесь `*` — это краткое обозначение «всех столбцов». [1] Таким образом, это равносильно записи:

```
SELECT city, temp_lo, temp_hi, prcp, date FROM weather;
```

В результате должно получиться:

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      43 |      57 |    0 | 1994-11-29
 Hayward       |      37 |      54 |      | 1994-11-29
(3 rows)
```

В списке выборки вы можете писать не только ссылки на столбцы, но и выражения. Например, вы можете написать:

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

И получить в результате:

```
     city      | temp_avg |    date
---------------+----------+------------
 San Francisco |       48 | 1994-11-27
 San Francisco |       50 | 1994-11-29
 Hayward       |       45 | 1994-11-29
(3 rows)
```

Обратите внимание, как предложение `AS` позволяет переименовать выходной столбец. (Само слово `AS` можно опускать.)

Запрос можно дополнить «условием», добавив предложение `WHERE`, ограничивающее множество возвращаемых строк. В предложении `WHERE` указывается логическое выражение (проверка истинности), которое служит фильтром строк: в результате оказываются только те строки, для которых это выражение истинно. В этом выражении могут присутствовать обычные логические операторы (`AND`, `OR` и `NOT`). Например, следующий запрос покажет, какая погода была в Сан-Франциско в дождливые дни:

[1] Хотя запросы `SELECT *` часто пишут экспромтом, это считается плохим стилем в производственном коде, так как результат таких запросов будет меняться при добавлении новых столбцов.

```
SELECT * FROM weather
    WHERE city = 'San Francisco' AND prcp > 0.0;
```

Результат:

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
(1 row)
```

Вы можете получить результаты запроса в определённом порядке:

```
SELECT * FROM weather
    ORDER BY city;
```

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 Hayward       |      37 |      54 |      | 1994-11-29
 San Francisco |      43 |      57 |    0 | 1994-11-29
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
```

В этом примере порядок сортировки определён не полностью, поэтому вы можете получить строки Сан-Франциско в любом порядке. Но вы всегда получите результат, показанный выше, если напишете:

```
SELECT * FROM weather
    ORDER BY city, temp_lo;
```

Если требуется, вы можете убрать дублирующиеся строки из результата запроса:

```
SELECT DISTINCT city
    FROM weather;
```

```
     city
---------------
 Hayward
 San Francisco
(2 rows)
```

И здесь порядок строк также может варьироваться. Чтобы получать неизменные результаты, соедините предложения `DISTINCT` и `ORDER BY`: [2]

```
SELECT DISTINCT city
    FROM weather
    ORDER BY city;
```

2.6. Соединения таблиц

До этого все наши запросы обращались только к одной таблице. Однако запросы могут также обращаться сразу к нескольким таблицам или обращаться к той же таблице так, что одновременно будут обрабатываться разные наборы её строк. Запрос, обращающийся к разным наборам строк одной или нескольких таблиц, называется *соединением* (JOIN). Например, мы захотели перечислить все погодные события вместе с координатами соответствующих городов. Для этого мы должны сравнить столбец `city` каждой строки таблицы `weather` со столбцом `name` всех строк таблицы `cities` и выбрать пары строк, для которых эти значения совпадают.

> **Примечание**
>
> Это не совсем точная модель. Обычно соединения выполняются эффективнее (сравниваются не все возможные пары строк), но это скрыто от пользователя.

[2] В некоторых СУБД, включая старые версии Postgres Pro, реализация предложения `DISTINCT` автоматически упорядочивает строки, так что `ORDER BY` добавлять не обязательно. Но стандарт SQL этого не требует и текущая версия Postgres Pro не гарантирует определённого порядка строк после `DISTINCT`.

Это можно сделать с помощью следующего запроса:

```
SELECT *
    FROM weather, cities
    WHERE city = name;
```

```
     city     |temp_lo|temp_hi| prcp|    date    |     name     | location
--------------+-------+-------+-----+------------+--------------+----------
 San Francisco|     46|     50| 0.25| 1994-11-27| San Francisco| (-194,53)
 San Francisco|     43|     57|    0| 1994-11-29| San Francisco| (-194,53)
(2 rows)
```

Обратите внимание на две особенности полученных данных:

- В результате нет строки с городом Хейуорд (Hayward). Так получилось потому, что в таблице `cities` нет строки для данного города, а при соединении все строки таблицы `weather`, для которых не нашлось соответствие, опускаются. Вскоре мы увидим, как это можно исправить.
- Название города оказалось в двух столбцах. Это правильно и объясняется тем, что столбцы таблиц `weather` и `cities` были объединены. Хотя на практике это нежелательно, поэтому лучше перечислить нужные столбцы явно, а не использовать `*`:

  ```
  SELECT city, temp_lo, temp_hi, prcp, date, location
      FROM weather, cities
      WHERE city = name;
  ```

Упражнение: Попробуйте определить, что будет делать этот запрос без предложения `WHERE`.

Так как все столбцы имеют разные имена, анализатор запроса автоматически понимает, к какой таблице они относятся. Если бы имена столбцов в двух таблицах повторялись, вам пришлось бы *дополнить* имена столбцов, конкретизируя, что именно вы имели в виду:

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.location
    FROM weather, cities
    WHERE cities.name = weather.city;
```

Вообще хорошим стилем считается указывать полные имена столбцов в запросе соединения, чтобы запрос не поломался, если позже в таблицы будут добавлены столбцы с повторяющимися именами.

Запросы соединения, которые вы видели до этого, можно также записать в другом виде:

```
SELECT *
    FROM weather INNER JOIN cities ON (weather.city = cities.name);
```

Эта запись не так распространена, как первый вариант, но мы показываем её, чтобы вам было проще понять следующие темы.

Сейчас мы выясним, как вернуть записи о погоде в городе Хейуорд. Мы хотим, чтобы запрос просканировал таблицу `weather` и для каждой её строки нашёл соответствующую строку в таблице `cities`. Если же такая строка не будет найдена, мы хотим, чтобы вместо значений столбцов из таблицы `cities` были подставлены «пустые значения». Запросы такого типа называются *внешними соединениями*. (Соединения, которые мы видели до этого, называются внутренними.) Эта команда будет выглядеть так:

```
SELECT *
    FROM weather LEFT OUTER JOIN cities ON (weather.city = cities.name);
```

```
     city     |temp_lo|temp_hi| prcp|    date    |     name     | location
--------------+-------+-------+-----+------------+--------------+----------
 Hayward      |     37|     54|     | 1994-11-29|              |
 San Francisco|     46|     50| 0.25| 1994-11-27| San Francisco| (-194,53)
 San Francisco|     43|     57|    0| 1994-11-29| San Francisco| (-194,53)
(3 rows)
```

Этот запрос называется *левым внешним соединением*, потому что из таблицы в левой части оператора будут выбраны все строки, а из таблицы справа только те, которые удалось сопоставить каким-нибудь строкам из левой. При выводе строк левой таблицы, для которых не удалось найти соответствия в правой, вместо столбцов правой таблицы подставляются пустые значения (NULL).

Упражнение: Существуют также правые внешние соединения и полные внешние соединения. Попробуйте выяснить, что они собой представляют.

В соединении мы также можем замкнуть таблицу на себя. Это называется *замкнутым соединением*. Например, представьте, что мы хотим найти все записи погоды, в которых температура лежит в диапазоне температур других записей. Для этого мы должны сравнить столбцы `temp_lo` и `temp_hi` каждой строки таблицы `weather` со столбцами `temp_lo` и `temp_hi` другого набора строк `weather`. Это можно сделать с помощью следующего запроса:

```
SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
    W2.city, W2.temp_lo AS low, W2.temp_hi AS high
    FROM weather W1, weather W2
    WHERE W1.temp_lo < W2.temp_lo
    AND W1.temp_hi > W2.temp_hi;

     city      | low | high |     city      | low | high
---------------+-----+------+---------------+-----+------
 San Francisco |  43 |   57 | San Francisco |  46 |   50
 Hayward       |  37 |   54 | San Francisco |  46 |   50
(2 rows)
```

Здесь мы ввели новые обозначения таблицы weather: `W1` и `W2`, чтобы можно было различить левую и правую стороны соединения. Вы можете использовать подобные псевдонимы и в других запросах для сокращения:

```
SELECT *
    FROM weather w, cities c
    WHERE w.city = c.name;
```

Вы будете встречать сокращения такого рода довольно часто.

2.7. Агрегатные функции

Как большинство других серверов реляционных баз данных, Postgres Pro поддерживает *агрегатные функции*. Агрегатная функция вычисляет единственное значение, обрабатывая множество строк. Например, есть агрегатные функции, вычисляющие: `count` (количество), `sum` (сумму), `avg` (среднее), `max` (максимум) и `min` (минимум) для набора строк.

К примеру, мы можем найти самую высокую из всех минимальных дневных температур:

```
SELECT max(temp_lo) FROM weather;

 max
-----
  46
(1 row)
```

Если мы хотим узнать, в каком городе (или городах) наблюдалась эта температура, можно попробовать:

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo);     НЕВЕРНО
```

но это не будет работать, так как агрегатную функцию `max` нельзя использовать в предложении `WHERE`. (Это ограничение объясняется тем, что предложение `WHERE` должно определить, для каких строк вычислять агрегатную функцию, так что оно, очевидно, должно вычисляться до агрегатных функций.) Однако как часто бывает, запрос можно перезапустить и получить желаемый результат, применив *подзапрос*:

```
SELECT city FROM weather
    WHERE temp_lo = (SELECT max(temp_lo) FROM weather);

     city
---------------
 San Francisco
(1 row)
```

Теперь всё в порядке — подзапрос выполняется отдельно и результат агрегатной функции вычисляется вне зависимости от того, что происходит во внешнем запросе.

Агрегатные функции также очень полезны в сочетании с предложением `GROUP BY`. Например, мы можем получить максимум минимальной дневной температуры в разрезе городов:

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city;

     city      | max
---------------+-----
 Hayward       |  37
 San Francisco |  46
(2 rows)
```

Здесь мы получаем по одной строке для каждого города. Каждый агрегатный результат вычисляется по строкам таблицы, соответствующим отдельному городу. Мы можем отфильтровать группируемые строки, используя `HAVING`, и подсчитываемые строки, используя `FILTER`:

```
SELECT city, max(temp_lo), count(*) FILTER (WHERE temp_lo < 30)
    FROM weather
    GROUP BY city
    HAVING max(temp_lo) < 40;

  city   | max | count
---------+-----+-------
 Hayward |  37 |     5
(1 row)
```

Мы получаем те же результаты, но только для тех городов, где все значения `temp_lo` меньше 40. Наконец, если нас интересуют только города, названия которых начинаются с «`S`», мы можем сделать:

```
SELECT city, max(temp_lo), count(*) FILTER (WHERE temp_lo < 30)
    FROM weather
    WHERE city LIKE 'S%'            -- 1
    GROUP BY city
    HAVING max(temp_lo) < 40;
```

1 Оператор `LIKE` (выполняющий сравнение по шаблону) рассматривается в Разделе 9.7.

Важно понимать, как соотносятся агрегатные функции и SQL-предложения `WHERE` и `HAVING`. Основное отличие `WHERE` от `HAVING` заключается в том, что `WHERE` сначала выбирает строки, а затем группирует их и вычисляет агрегатные функции (таким образом, она отбирает строки для вычисления агрегатов), тогда как `HAVING` отбирает строки групп после группировки и вычисления агрегатных функций. Как следствие, предложение `WHERE` не должно содержать агрегатных функций; не имеет смысла использовать агрегатные функции для определения строк для вычисления агрегатных функций. Предложение `HAVING`, напротив, всегда содержит агрегатные функции. (Строго говоря, вы можете написать предложение `HAVING`, не используя агрегаты, но это редко бывает полезно. То же самое условие может работать более эффективно на стадии `WHERE`.)

В предыдущем примере мы смогли применить фильтр по названию города в предложении `WHERE`, так как названия не нужно агрегировать. Такой фильтр эффективнее, чем дополнительное

ограничение HAVING, потому что с ним не приходится группировать и вычислять агрегаты для всех строк, не удовлетворяющих условию WHERE.

2.8. Изменение данных

Данные в существующих строках можно изменять, используя команду UPDATE. Например, предположим, что вы обнаружили, что все значения температуры после 28 ноября завышены на два градуса. Вы можете поправить ваши данные следующим образом:

```
UPDATE weather
    SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
    WHERE date > '1994-11-28';
```

Посмотрите на новое состояние данных:

```
SELECT * FROM weather;

     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
 Hayward       |      35 |      52 |      | 1994-11-29
(3 rows)
```

2.9. Удаление данных

Строки также можно удалить из таблицы, используя команду DELETE. Предположим, что вас больше не интересует погода в Хейуорде. В этом случае вы можете удалить ненужные строки из таблицы:

```
DELETE FROM weather WHERE city = 'Hayward';
```

Записи всех наблюдений, относящиеся к Хейуорду, удалены.

```
SELECT * FROM weather;

     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
(2 rows)
```

Остерегайтесь операторов вида

```
DELETE FROM имя_таблицы;
```

Без указания условия DELETE удалит *все* строки данной таблицы, полностью очистит её. При этом система не попросит вас подтвердить операцию!

Глава 3. Расширенные возможности

3.1. Введение

В предыдущей главе мы изучили азы использования SQL для хранения и обработки данных в Postgres Pro. Теперь мы обсудим более сложные возможности SQL, помогающие управлять данными и предотвратить их потерю или порчу. В конце главы мы рассмотрим некоторые расширения Postgres Pro.

В этой главе мы будем время от времени ссылаться на примеры, приведённые в Главе 2 и изменять или развивать их, поэтому будет полезно сначала прочитать предыдущую главу. Некоторые примеры этой главы также можно найти в файле `advanced.sql` в каталоге tutorial. Кроме того, этот файл содержит пример данных для загрузки (здесь она повторно не рассматривается). Если вы не знаете, как использовать этот файл, обратитесь к Разделу 2.1.

3.2. Представления

Вспомните запросы, с которыми мы имели дело в Разделе 2.6. Предположим, что вас интересует составной список из погодных записей и координат городов, но вы не хотите каждый раз вводить весь этот запрос. Вы можете создать *представление* по данному запросу, фактически присвоить имя запросу, а затем обращаться к нему как к обычной таблице:

```
CREATE VIEW myview AS
    SELECT name, temp_lo, temp_hi, prcp, date, location
        FROM weather, cities
        WHERE city = name;

SELECT * FROM myview;
```

Активное использование представлений — это ключевой аспект хорошего проектирования баз данных SQL. Представления позволяют вам скрыть внутреннее устройство ваших таблиц, которые могут меняться по мере развития приложения, за надёжными интерфейсами.

Представления можно использовать практически везде, где можно использовать обычные таблицы. И довольно часто представления создаются на базе других представлений.

3.3. Внешние ключи

Вспомните таблицы `weather` и `cities` из Главы 2. Давайте рассмотрим следующую задачу: вы хотите добиться, чтобы никто не мог вставить в таблицу `weather` строки, для которых не находится соответствующая строка в таблице `cities`. Это называется обеспечением *ссылочной целостности* данных. В простых СУБД это пришлось бы реализовать (если это вообще возможно) так: сначала явно проверить, есть ли соответствующие записи в таблице `cities`, а затем отклонить или вставить новые записи в таблицу `weather`. Этот подход очень проблематичен и неудобен, поэтому всё это Postgres Pro может сделать за вас.

Новое объявление таблицы будет выглядеть так:

```
CREATE TABLE cities (
        name     varchar(80) primary key,
        location point
);

CREATE TABLE weather (
        city      varchar(80) references cities(name),
        temp_lo   int,
        temp_hi   int,
        prcp      real,
        date      date
```

```
);
```

Теперь попробуйте вставить недопустимую запись:

```
INSERT INTO weather VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');
```

```
ОШИБКА:  INSERT или UPDATE в таблице "weather" нарушает ограничение внешнего
  ключа "weather_city_fkey"
ПОДРОБНОСТИ:  Ключ (city)=(Berkeley) отсутствует в таблице "cities".
```

Поведение внешних ключей можно подстроить согласно требованиям вашего приложения. Мы не будем усложнять этот простой пример в данном введении, но вы можете обратиться за дополнительной информацией к Главе 5. Правильно применяя внешние ключи, вы определённо создадите более качественные приложения, поэтому мы настоятельно рекомендуем изучить их.

3.4. Транзакции

Транзакции — это фундаментальное понятие во всех СУБД. Суть транзакции в том, что она объединяет последовательность действий в одну операцию «всё или ничего». Промежуточные состояния внутри последовательности не видны другим транзакциям, и если что-то помешает успешно завершить транзакцию, ни один из результатов этих действий не сохранится в базе данных.

Например, рассмотрим базу данных банка, в которой содержится информация о счетах клиентов, а также общие суммы по отделениям банка. Предположим, что мы хотим перевести 100 долларов со счёта Алисы на счёт Боба. Простоты ради, соответствующие SQL-команды можно записать так:

```
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
```

Точное содержание команд здесь не важно, важно лишь то, что для выполнения этой довольно простой операции потребовалось несколько отдельных действий. При этом с точки зрения банка необходимо, чтобы все эти действия выполнились вместе, либо не выполнились совсем. Если Боб получит 100 долларов, но они не будут списаны со счёта Алисы, объяснить это сбоем системы определённо не удастся. И наоборот, Алиса вряд ли будет довольна, если она переведёт деньги, а до Боба они не дойдут. Нам нужна гарантия, что если что-то помешает выполнить операцию до конца, ни одно из действий не оставит следа в базе данных. И мы получаем эту гарантию, объединяя действия в одну *транзакцию*. Говорят, что транзакция *атомарна*: с точки зрения других транзакций она либо выполняется и фиксируется полностью, либо не фиксируется совсем.

Нам также нужна гарантия, что после завершения и подтверждения транзакции системой баз данных, её результаты в самом деле сохраняются и не будут потеряны, даже если вскоре произойдёт авария. Например, если мы списали сумму и выдали её Бобу, мы должны исключить возможность того, что сумма на его счёте восстановится, как только он выйдет за двери банка. Транзакционная база данных гарантирует, что все изменения записываются в постоянное хранилище (например, на диск) до того, как транзакция будет считаться завершённой.

Другая важная характеристика транзакционных баз данных тесно связана с атомарностью изменений: когда одновременно выполняется множество транзакций, каждая из них не видит незавершённые изменения, произведённые другими. Например, если одна транзакция подсчитывает баланс по отделениям, будет неправильно, если она посчитает расход в отделении Алисы, но не учтёт приход в отделении Боба, или наоборот. Поэтому свойство транзакций «всё или ничего» должно определять не только, как изменения сохраняются в базе данных, но и как они видны в процессе работы. Изменения, производимые открытой транзакцией, невидимы для других транзакций, пока она не будет завершена, а затем они становятся видны все сразу.

В Postgres Pro транзакция определяется набором SQL-команд, окружённым командами BEGIN и COMMIT. Таким образом, наша банковская транзакция должна была бы выглядеть так:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
-- ...
COMMIT;
```

Если в процессе выполнения транзакции мы решим, что не хотим фиксировать её изменения (например, потому что оказалось, что баланс Алисы стал отрицательным), мы можем выполнить команду ROLLBACK вместо COMMIT, и все наши изменения будут отменены.

Postgres Pro на самом деле отрабатывает каждый SQL-оператор как транзакцию. Если вы не вставите команду BEGIN, то каждый отдельный оператор будет неявно окружён командами BEGIN и COMMIT (в случае успешного завершения). Группу операторов, окружённых командами BEGIN и COMMIT иногда называют *блоком транзакции*.

> **Примечание**
>
> Некоторые клиентские библиотеки добавляют команды BEGIN и COMMIT автоматически и неявно создают за вас блоки транзакций. Подробнее об этом вы можете узнать в документации интересующего вас интерфейса.

Операторами в транзакции можно также управлять на более детальном уровне, используя *точки сохранения*. Точки сохранения позволяют выборочно отменять некоторые части транзакции и фиксировать все остальные. Определив точку сохранения с помощью SAVEPOINT, при необходимости вы можете вернуться к ней с помощью команды ROLLBACK TO. Все изменения в базе данных, произошедшие после точки сохранения и до момента отката, отменяются, но изменения, произведённые ранее, сохраняются.

Когда вы возвращаетесь к точке сохранения, она продолжает существовать, так что вы можете откатываться к ней несколько раз. С другой стороны, если вы уверены, что вам не придётся откатываться к определённой точке сохранения, её можно удалить, чтобы система высвободила ресурсы. Помните, что при удалении или откате к точке сохранения все точки сохранения, определённые после неё, автоматически уничтожаются.

Всё это происходит в блоке транзакции, так что в других сеансах работы с базой данных этого не видно. Совершённые действия становятся видны для других сеансов все сразу, только когда вы фиксируете транзакцию, а отменённые действия не видны вообще никогда.

Вернувшись к банковской базе данных, предположим, что мы списываем 100 долларов со счёта Алисы, добавляем их на счёт Боба, и вдруг оказывается, что деньги нужно было перевести Уолли. В данном случае мы можем применить точки сохранения:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- ошибочное действие... забыть его и использовать счёт Уолли
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

Этот пример, конечно, несколько надуман, но он показывает, как можно управлять выполнением команд в блоке транзакций, используя точки сохранения. Более того, ROLLBACK TO — это

единственный способ вернуть контроль над блоком транзакций, оказавшимся в прерванном состоянии из-за ошибки системы, не считая возможности полностью отменить её и начать снова.

3.5. Оконные функции

Оконная функция выполняет вычисления для набора строк, некоторым образом связанных с текущей строкой. Её действие можно сравнить с вычислением, производимым агрегатной функцией. Однако с оконными функциями строки не группируются в одну выходную строку, что имеет место с обычными, не оконными, агрегатными функциями. Вместо этого, эти строки остаются отдельными сущностями. Внутри же, оконная функция, как и агрегатная, может обращаться не только к текущей строке результата запроса.

Вот пример, показывающий, как сравнить зарплату каждого сотрудника со средней зарплатой его отдела:

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname)
  FROM empsalary;
```

```
  depname  | empno | salary |          avg
-----------+-------+--------+-----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     9 |   4500 | 5020.0000000000000000
 develop   |     8 |   6000 | 5020.0000000000000000
 develop   |    10 |   5200 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 personnel |     2 |   3900 | 3700.0000000000000000
 sales     |     3 |   4800 | 4866.6666666666666667
 sales     |     1 |   5000 | 4866.6666666666666667
 sales     |     4 |   4800 | 4866.6666666666666667
(10 rows)
```

Первые три столбца извлекаются непосредственно из таблицы `empsalary`, при этом для каждой строки таблицы есть строка результата. В четвёртом столбце оказалось среднее значение, вычисленное по всем строкам, имеющим то же значение `depname`, что и текущая строка. (Фактически среднее вычисляет та же обычная, не оконная функция `avg`, но предложение `OVER` превращает её в оконную, так что её действие ограничивается рамками окон.)

Вызов оконной функции всегда содержит предложение `OVER`, следующее за названием и аргументами оконной функции. Это синтаксически отличает её от обычной, не оконной агрегатной функции. Предложение `OVER` определяет, как именно нужно разделить строки запроса для обработки оконной функцией. Предложение `PARTITION BY`, дополняющее `OVER`, разделяет строки по группам, или разделам, объединяя одинаковые значения выражений `PARTITION BY`. Оконная функция вычисляется по строкам, попадающим в один раздел с текущей строкой.

Вы можете также определять порядок, в котором строки будут обрабатываться оконными функциями, используя `ORDER BY` в `OVER`. (Порядок `ORDER BY` для окна может даже не совпадать с порядком, в котором выводятся строки.) Например:

```
SELECT depname, empno, salary,
       rank() OVER (PARTITION BY depname ORDER BY salary DESC)
FROM empsalary;
```

```
  depname  | empno | salary | rank
-----------+-------+--------+------
 develop   |     8 |   6000 |    1
 develop   |    10 |   5200 |    2
 develop   |    11 |   5200 |    2
 develop   |     9 |   4500 |    4
 develop   |     7 |   4200 |    5
```

```
 personnel |      2 |   3900 |    1
 personnel |      5 |   3500 |    2
 sales     |      1 |   5000 |    1
 sales     |      4 |   4800 |    2
 sales     |      3 |   4800 |    2
(10 rows)
```

Как показано здесь, функция `rank` выдаёт порядковый номер для каждого уникального значения в разделе текущей строки, по которому выполняет сортировку предложение `ORDER BY`. У функции `rank` нет параметров, так как её поведение полностью определяется предложением `OVER`.

Строки, обрабатываемые оконной функцией, представляют собой «виртуальные таблицы», созданные из предложения `FROM` и затем прошедшие через фильтрацию и группировку `WHERE` и `GROUP BY` и, возможно, условие `HAVING`. Например, строка, отфильтрованная из-за нарушения условия `WHERE`, не будет видна для оконных функций. Запрос может содержать несколько оконных функций, разделяющих данные по-разному с применением разных предложений `OVER`, но все они будут обрабатывать один и тот же набор строк этой виртуальной таблицы.

Мы уже видели, что `ORDER BY` можно опустить, если порядок строк не важен. Также возможно опустить `PARTITION BY`, в этом случае образуется один раздел, содержащий все строки.

Есть ещё одно важное понятие, связанное с оконными функциями: для каждой строки существует набор строк в её разделе, называемый *рамкой окна*. Некоторые оконные функции обрабатывают только строки рамки окна, а не всего раздела. По умолчанию с указанием `ORDER BY` рамка состоит из всех строк от начала раздела до текущей строки и строк, равных текущей по значению выражения `ORDER BY`. Без `ORDER BY` рамка по умолчанию состоит из всех строк раздела. [1] Посмотрите на пример использования `sum`:

```
SELECT salary, sum(salary) OVER () FROM empsalary;
```

```
 salary |  sum
--------+-------
   5200 | 47100
   5000 | 47100
   3500 | 47100
   4800 | 47100
   3900 | 47100
   4200 | 47100
   4500 | 47100
   4800 | 47100
   6000 | 47100
   5200 | 47100
(10 rows)
```

Так как в этом примере нет указания `ORDER BY` в предложении `OVER`, рамка окна содержит все строки раздела, а он, в свою очередь, без предложения `PARTITION BY` включает все строки таблицы; другими словами, сумма вычисляется по всей таблице и мы получаем один результат для каждой строки результата. Но если мы добавим `ORDER BY`, мы получим совсем другие результаты:

```
SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary;
```

```
 salary |  sum
--------+-------
   3500 |  3500
   3900 |  7400
   4200 | 11600
   4500 | 16100
   4800 | 25700
   4800 | 25700
```

[1] Рамки окна можно определять и другими способами, но в этом введении они не рассматриваются. Узнать о них подробнее вы можете в Подразделе 4.2.8.

```
   5000 | 30700
   5200 | 41100
   5200 | 41100
   6000 | 47100
(10 rows)
```

Здесь в сумме накапливаются зарплаты от первой (самой низкой) до текущей, включая повторяющиеся текущие значения (обратите внимание на результат в строках с одинаковой зарплатой).

Оконные функции разрешается использовать в запросе только в списке `SELECT` и предложении `ORDER BY`. Во всех остальных предложениях, включая `GROUP BY`, `HAVING` и `WHERE`, они запрещены. Это объясняется тем, что логически они выполняются после этих предложений, а также после не оконных агрегатных функций, и значит агрегатную функцию можно вызывать в аргументах оконной, но не наоборот.

Если вам нужно отфильтровать или сгруппировать строки после вычисления оконных функций, вы можете использовать вложенный запрос. Например:

```
SELECT depname, empno, salary, enroll_date
FROM
  (SELECT depname, empno, salary, enroll_date,
    rank() OVER (PARTITION BY depname ORDER BY salary DESC, empno) AS pos
   FROM empsalary
  ) AS ss
WHERE pos < 3;
```

Данный запрос покажет только те строки внутреннего запроса, у которых `rank` (порядковый номер) меньше 3.

Когда в запросе вычисляются несколько оконных функций для одинаково определённых окон, конечно можно написать для каждой из них отдельное предложение `OVER`, но при этом оно будет дублироваться, что неизбежно будет провоцировать ошибки. Поэтому лучше определение окна выделить в предложение `WINDOW`, а затем ссылаться на него в `OVER`. Например:

```
SELECT sum(salary) OVER w, avg(salary) OVER w
  FROM empsalary
  WINDOW w AS (PARTITION BY depname ORDER BY salary DESC);
```

Подробнее об оконных функциях можно узнать в Подразделе 4.2.8, Разделе 9.21, Подразделе 7.2.5 и в справке SELECT.

3.6. Наследование

Наследование — это концепция, взятая из объектно-ориентированных баз данных. Она открывает множество интересных возможностей при проектировании баз данных.

Давайте создадим две таблицы: `cities` (города) и `capitals` (столицы штатов). Естественно, столицы штатов также являются городами, поэтому нам нужно явным образом добавлять их в результат, когда мы хотим просмотреть все города. Если вы проявите смекалку, вы можете предложить, например, такое решение:

```
CREATE TABLE capitals (
  name       text,
  population real,
  elevation  int,    -- (высота в футах)
  state      char(2)
);

CREATE TABLE non_capitals (
  name       text,
```

```
  population real,
  elevation  int     -- (высота в футах)
);

CREATE VIEW cities AS
  SELECT name, population, elevation FROM capitals
    UNION
  SELECT name, population, elevation FROM non_capitals;
```

Оно может устраивать, пока мы извлекаем данные, но если нам потребуется изменить несколько строк, это будет выглядеть некрасиво.

Поэтому есть лучшее решение:

```
CREATE TABLE cities (
  name       text,
  population real,
  elevation  int     -- (высота в футах)
);

CREATE TABLE capitals (
  state      char(2) UNIQUE NOT NULL
) INHERITS (cities);
```

В данном случае строка таблицы `capitals` *наследует* все столбцы (`name`, `population` и `elevation`) от *родительской таблицы* `cities`. Столбец `name` имеет тип `text`, собственный тип Postgres Pro для текстовых строк переменной длины. А в таблицу `capitals` добавлен дополнительный столбец `state`, в котором будет указан буквенный код штата. В Postgres Pro таблица может наследоваться от ноля или нескольких других таблиц.

Например, следующий запрос выведет названия всех городов, включая столицы, находящихся выше 500 футов над уровнем моря:

```
SELECT name, elevation
  FROM cities
  WHERE elevation > 500;
```

Результат его выполнения:

```
   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
 Madison   |       845
(3 rows)
```

А следующий запрос находит все города, которые не являются столицами штатов, но также находятся выше 500 футов:

```
SELECT name, elevation
    FROM ONLY cities
    WHERE elevation > 500;
```

```
   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
(2 rows)
```

Здесь слово `ONLY` перед названием таблицы `cities` указывает, что запрос следует выполнять только для строк таблицы `cities`, не включая таблицы, унаследованные от `cities`. Многие операторы, которые мы уже обсудили — `SELECT`, `UPDATE` и `DELETE` — поддерживают указание `ONLY`.

> **Примечание**
>
> Хотя наследование часто бывает полезно, оно не интегрировано с ограничениями уникальности и внешними ключами, что ограничивает его применимость. Подробнее это описывается в Разделе 5.9.

3.7. Заключение

Postgres Pro имеет множество возможностей, не затронутых в этом кратком введении, рассчитанном на начинающих пользователей SQL. Эти возможности будут рассмотрены в деталях в продолжении книги.

Если вам необходима дополнительная вводная информация, посетите *сайт PostgreSQL*, там вы найдёте ссылки на другие ресурсы.

Часть II. Язык SQL

В этой части книги описывается использование языка SQL в Postgres Pro. Мы начнём с описания общего синтаксиса SQL, затем расскажем, как создавать структуры для хранения данных, как наполнять базу данных и как выполнять запросы к ней. В продолжении будут перечислены существующие типы данных и функции, применяемые с командами SQL. И наконец, закончится эта часть рассмотрением важных аспектов настройки базы данных для оптимальной производительности.

Материал этой части упорядочен так, чтобы новичок мог прочитать её от начала до конца и полностью понять все темы, не забегая вперёд. При этом главы сделаны самодостаточными, так что опытные пользователи могут читать главы по отдельности. Информация в этой части книги представлена в повествовательном стиле и разделена по темам. Если же вас интересует формальное и полное описание определённой команды, см. Часть VI.

Читатели этой части книги должны уже знать, как подключаться к базе данных Postgres Pro и выполнять команды SQL. Если вы ещё не знаете этого, рекомендуется сначала прочитать Часть I. Команды SQL обычно вводятся в psql — интерактивном терминальном приложении Postgres Pro, но можно воспользоваться и другими программами с подобными функциями.

Глава 4. Синтаксис SQL

В этой главе описывается синтаксис языка SQL. Тем самым закладывается фундамент для следующих глав, где будет подробно рассмотрено, как с помощью команд SQL описывать и изменять данные.

Мы советуем прочитать эту главу и тем, кто уже знаком SQL, так как в ней описываются несколько правил и концепций, которые реализованы в разных базах данных SQL по-разному или относятся только к Postgres Pro.

4.1. Лексическая структура

SQL-программа состоит из последовательности *команд*. Команда, в свою очередь, представляет собой последовательность *компонентов*, оканчивающуюся точкой с запятой («;»). Конец входного потока также считается концом команды. Какие именно компоненты допустимы для конкретной команды, зависит от её синтаксиса.

Компонентом команды может быть *ключевое слово*, *идентификатор*, *идентификатор в кавычках*, *строка* (или константа) или специальный символ. Компоненты обычно разделяются пробельными символами (пробел, табуляция, перевод строки), но это не требуется, если нет неоднозначности (например, когда спецсимвол оказывается рядом с компонентом другого типа).

Например, следующий текст является правильной (синтаксически) SQL-программой:

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

Это последовательность трёх команд, по одной в строке (хотя их можно было разместить и в одну строку или наоборот, разделить команды на несколько строк).

Кроме этого, SQL-программы могут содержать *комментарии*. Они не являются компонентами команд, а по сути равносильны пробельным символам.

Синтаксис SQL не очень строго определяет, какие компоненты идентифицируют команды, а какие — их операнды или параметры. Первые несколько компонентов обычно содержат имя команды, так что в данном примере мы можем говорить о командах «SELECT», «UPDATE» и «INSERT». Но например, команда `UPDATE` требует, чтобы также в определённом положении всегда стоял компонент `SET`, а `INSERT` в приведённом виде требует наличия компонента `VALUES`. Точные синтаксические правила для каждой команды описаны в Части VI.

4.1.1. Идентификаторы и ключевые слова

Показанные выше команды содержали компоненты `SELECT`, `UPDATE` и `VALUES`, которые являются примерами *ключевых слов*, то есть слов, имеющих фиксированное значение в языке SQL. Компоненты `MY_TABLE` и `A` являются примерами *идентификаторов*. Они идентифицируют имена таблиц, столбцов или других объектов баз данных, в зависимости от того, где они используются. Поэтому иногда их называют просто «именами». Ключевые слова и идентификаторы имеют одинаковую лексическую структуру, то есть, не зная языка, нельзя определить, является ли некоторый компонент ключевым словом или идентификатором. Полный список ключевых слов приведён в Приложении C.

Идентификаторы и ключевые слова SQL должны начинаться с буквы (`a-z`, хотя допускаются также не латинские буквы и буквы с диакритическими знаками) или подчёркивания (_). Последующими символами в идентификаторе или ключевом слове могут быть буквы, цифры (`0-9`), знаки доллара (`$`) или подчёркивания. Заметьте, что строго следуя букве стандарта SQL, знаки доллара нельзя использовать в идентификаторах, так что их использование вредит переносимости приложений. В стандарте SQL гарантированно не будет ключевых слов с цифрами и начинающихся или заканчивающихся подчёркиванием, так что идентификаторы такого вида защищены от возможных конфликтов с будущими расширениями стандарта.

Система выделяет для идентификатора не более NAMEDATALEN-1 байт, а более длинные имена усекаются. По умолчанию NAMEDATALEN равно 64, так что максимальная длина идентификатора равна 63 байтам. Если этого недостаточно, этот предел можно увеличить, изменив константу NAMEDATALEN в файле src/include/pg_config_manual.h.

Ключевые слова и идентификаторы без кавычек воспринимаются системой без учёта регистра. Таким образом:

```
UPDATE MY_TABLE SET A = 5;
```

равносильно записи:

```
uPDaTE my_TabLE SeT a = 5;
```

Часто используется неформальное соглашение записывать ключевые слова заглавными буквами, а имена строчными, например:

```
UPDATE my_table SET a = 5;
```

Есть и другой тип идентификаторов: *отделённые идентификаторы* или *идентификаторы в кавычках*. Они образуются при заключении обычного набора символов в двойные кавычки ("). Такие идентификаторы всегда будут считаться идентификаторами, но не ключевыми словами. Так "select" можно использовать для обозначения столбца или таблицы «select», тогда как select без кавычек будет воспринят как ключевое слово и приведёт к ошибке разбора команды в месте, где ожидается имя таблицы или столбца. Тот же пример можно переписать с идентификаторами в кавычках следующим образом:

```
UPDATE "my_table" SET "a" = 5;
```

Идентификаторы в кавычках могут содержать любые символы, за исключением символа с кодом 0. (Чтобы включить в такой идентификатор кавычки, продублируйте их.) Это позволяет создавать таблицы и столбцы с именами, которые иначе были бы невозможны, например, с пробелами или амперсандами. Ограничение длины при этом сохраняется.

Ещё один вариант идентификаторов в кавычках позволяет использовать символы Unicode по их кодам. Такой идентификатор начинается с U& (строчная или заглавная U и амперсанд), а затем сразу без пробелов идёт двойная кавычка, например U&"foo". (Заметьте, что при этом возникает неоднозначность с оператором &. Чтобы её избежать, окружайте этот оператор пробелами.) Затем в кавычках можно записывать символы Unicode двумя способами: обратная косая черта, а за ней код символа из четырёх шестнадцатеричных цифр, либо обратная косая черта, знак плюс, а затем код из шести шестнадцатеричных цифр. Например, идентификатор "data" можно записать так:

```
U&"d\0061t\+000061"
```

В следующем менее тривиальном примере закодировано русское слово «слон», записанное кириллицей:

```
U&"\0441\043B\043E\043D"
```

Если вы хотите использовать не обратную косую черту, а другой спецсимвол, его можно указать, добавив UESCAPE после строки, например:

```
U&"d!0061t!+000061" UESCAPE '!'
```

В качестве спецсимвола можно выбрать любой символ, кроме шестнадцатеричной цифры, знака плюс, апострофа, кавычки или пробельного символа. Заметьте, что спецсимвол заключается не в двойные кавычки, а в апострофы.

Чтобы сделать спецсимволом знак апострофа, напишите его дважды.

Unicode-формат полностью поддерживается только при использовании на сервере кодировки UTF8. Когда используются другие кодировки, допускается указание только ASCII-символов (с кодами до \007F). И в четырёх, и в шестизначной форме можно записывать суррогатные пары UTF-16 и

таким образом составлять символы с кодами больше чем U+FFFF, хотя наличие шестизначной формы технически делает это ненужным. (Суррогатные пары не сохраняются непосредственно, а объединяются в один символ, который затем кодируется в UTF-8.)

Идентификатор, заключённый в кавычки, становится зависимым от регистра, тогда как идентификаторы без кавычек всегда переводятся в нижний регистр. Например, идентификаторы `FOO`, `foo` и `"foo"` считаются одинаковыми в Postgres Pro, но `"Foo"` и `"FOO"` отличны друг от друга и от предыдущих трёх. (Приведение имён без кавычек к нижнему регистру, как это делает Postgres Pro, несовместимо со стандартом SQL, который говорит о том, что имена должны приводиться к верхнему регистру. То есть, согласно стандарту `foo` должно быть эквивалентно `"FOO"`, а не `"foo"`. Поэтому при создании переносимых приложений рекомендуется либо всегда заключать определённое имя в кавычки, либо не заключать никогда.)

4.1.2. Константы

В Postgres Pro есть три типа констант *подразумеваемых типов*: строки, битовые строки и числа. Константы можно также записывать, указывая типы явно, что позволяет представить их более точно и обработать более эффективно. Эти варианты рассматриваются в следующих подразделах.

4.1.2.1. Строковые константы

Строковая константа в SQL — это обычная последовательность символов, заключённая в апострофы (`'`), например: `'Это строка'`. Чтобы включить апостроф в строку, напишите в ней два апострофа рядом, например: `'Жанна д''Арк'`. Заметьте, это *не* то же самое, что двойная кавычка (`"`).

Две строковые константы, разделённые пробельными символами *и минимум одним переводом строки*, объединяются в одну и обрабатываются, как если бы строка была записана в одной константе. Например:

```
SELECT 'foo'
'bar';
```

эквивалентно:

```
SELECT 'foobar';
```

но эта запись:

```
SELECT 'foo'      'bar';
```

считается синтаксической ошибкой. (Это несколько странное поведение определено в стандарте SQL, Postgres Pro просто следует ему.)

4.1.2.2. Строковые константы со спецпоследовательностями в стиле C

Postgres Pro также принимает «спецпоследовательности», что является расширением стандарта SQL. Строка со спецпоследовательностями начинается с буквы `E` (заглавной или строчной), стоящей непосредственно перед апострофом, например: `E'foo'`. (Когда константа со спецпоследовательностью разбивается на несколько строк, букву `E` нужно поставить только перед первым открывающим апострофом.) Внутри таких строк символ обратной косой черты (`\`) начинает C-подобные *спецпоследовательности*, в которых сочетание обратной косой черты со следующим символом(ами) даёт определённое байтовое значение, как показано в Таблице 4.1.

Таблица 4.1. Спецпоследовательности

Спецпоследовательность	Интерпретация
`\b`	символ «забой»
`\f`	подача формы
`\n`	новая строка
`\r`	возврат каретки

Спецпоследовательность	Интерпретация
`\t`	табуляция
`\o`, `\oo`, `\ooo` (*o* = 0 - 7)	восьмеричное значение байта
`\xh`, `\xhh` (*h* = 0 — 9, A — F)	шестнадцатеричное значение байта
`\uxxxx`, `\Uxxxxxxxx` (*x* = 0 — 9, A — F)	16- или 32-битный шестнадцатеричный код символа Unicode

Любой другой символ, идущий после обратной косой черты, воспринимается буквально. Таким образом, чтобы включить в строку обратную косую черту, нужно написать две косых черты (`\\`). Так же можно включить в строку апостроф, написав `\'`, в дополнение к обычному способу `''`.

Вы должны позаботиться, чтобы байтовые последовательности, которые вы создаёте таким образом, особенно в восьмеричной и шестнадцатеричной записи, образовывали допустимые символы в серверной кодировке. Когда сервер работает с кодировкой UTF-8, вместо такой записи байт следует использовать спецпоследовательности Unicode или альтернативный синтаксис Unicode, описанный в Подразделе 4.1.2.3. (В противном случае придётся кодировать символы UTF-8 вручную и выписывать их по байтам, что очень неудобно.)

Спецпоследовательности с Unicode полностью поддерживаются только при использовании на сервере кодировки `UTF8`. Когда используются другие кодировки, допускается указание только ASCII-символов (с кодами до `\u007F`). И в четырёх, и в восьмизначной форме можно записывать суррогатные пары UTF-16 и таким образом составлять символы с кодами больше чем U+FFFF, хотя наличие восьмизначной формы технически делает это ненужным. (Когда суррогатные пары используются с серверной кодировкой `UTF8`, они сначала объединяются в один символ, который затем кодируется в UTF-8.)

> **Внимание**
>
> Если параметр конфигурации standard_conforming_strings имеет значение `off`, Postgres Pro распознаёт обратную косую черту как спецсимвол и в обычных строках, и в строках со спецпоследовательностями. Однако в версии PostgreSQL 9.1 по умолчанию принято значение `on`, и в этом случае обратная косая черта распознаётся только в спецстроках. Это поведение больше соответствует стандарту, хотя может нарушить работу приложений, рассчитанных на предыдущий режим, когда обратная косая черта распознавалась везде. В качестве временного решения вы можете изменить этот параметр на `off`, но лучше уйти от такой практики. Если вам нужно, чтобы обратная косая черта представляла специальный символ, задайте строковую константу с `E`.
>
> В дополнение к `standard_conforming_strings` поведением обратной косой черты в строковых константах управляют параметры escape_string_warning и backslash_quote.

Строковая константа не может включать символ с кодом 0.

4.1.2.3. Строковые константы со спецпоследовательностями Unicode

Postgres Pro также поддерживает ещё один вариант спецпоследовательностей, позволяющий включать в строки символы Unicode по их кодам. Строковая константа со спецпоследовательностями Unicode начинается с `U&` (строчная или заглавная U и амперсанд), а затем сразу без пробелов идёт апостроф, например `U&'foo'`. (Заметьте, что при этом возникает неоднозначность с оператором `&`. Чтобы её избежать, окружайте этот оператор пробелами.) Затем в апострофах можно записывать символы Unicode двумя способами: обратная косая черта, а за ней код символа из четырёх шестнадцатеричных цифр, либо обратная косая черта, знак плюс, а затем код из шести шестнадцатеричных цифр. Например, строку `'data'` можно записать так:

```
U&'d\0061t\+000061'
```

В следующем менее тривиальном примере закодировано русское слово «слон», записанное кириллицей:

```
U&'\0441\043B\043E\043D'
```

Если вы хотите использовать не обратную косую черту, а другой спецсимвол, его можно указать, добавив `UESCAPE`после строки, например:

```
U&'d!0061t!+000061' UESCAPE '!'
```

В качестве спецсимвола можно выбрать любой символ, кроме шестнадцатеричной цифры, знака плюс, апострофа, кавычки или пробельного символа.

Спецпоследовательности с Unicode поддерживаются только при использовании на сервере кодировки `UTF8`. Когда используются другие кодировки, допускается указание только ASCII-символов (с кодами до `\007F`). И в четырёх, и в шестизначной форме можно записывать суррогатные пары UTF-16 и таким образом составлять символы с кодами больше чем U+FFFF, хотя наличие шестизначной формы технически делает это ненужным. (Когда суррогатные пары используются с серверной кодировкой `UTF8`, они сначала объединяются в один символ, который затем кодируется в UTF-8.)

Также заметьте, что спецпоследовательности Unicode в строковых константах работают, только когда параметр конфигурации standard_conforming_strings равен `on`. Это объясняется тем, что иначе клиентские программы, проверяющие SQL-операторы, можно будет ввести в заблуждение и эксплуатировать это как уязвимость, например, для SQL-инъекций. Если этот параметр имеет значение `off`, эти спецпоследовательности будут вызывать ошибку.

Чтобы включить спецсимвол в строку буквально, напишите его дважды.

4.1.2.4. Строковые константы, заключённые в доллары

Хотя стандартный синтаксис для строковых констант обычно достаточно удобен, он может плохо читаться, когда строка содержит много апострофов или обратных косых черт, так как каждый такой символ приходится дублировать. Чтобы и в таких случаях запросы оставались читаемыми, Postgres Pro предлагает ещё один способ записи строковых констант — «заключение строк в доллары». Строковая константа, заключённая в доллары, начинается со знака доллара (`$`), необязательного «тега» из нескольких символов и ещё одного знака доллара, затем содержит обычную последовательность символов, составляющую строку, и оканчивается знаком доллара, тем же тегом и замыкающим знаком доллара. Например, строку «Жанна д'Арк» можно записать в долларах двумя способами:

```
$$Жанна д'Арк$$
$SomeTag$Жанна д'Арк$SomeTag$
```

Заметьте, что внутри такой строки апострофы не нужно записывать особым образом. На самом деле, в строке, заключённой в доллары, все символы можно записывать в чистом виде: содержимое строки всегда записывается буквально. Ни обратная косая черта, ни даже знак доллара не являются спецсимволами, если только они не образуют последовательность, соответствующую открывающему тегу.

Строковые константы в долларах можно вкладывать друг в друга, выбирая на разных уровнях вложенности разные теги. Чаще всего это используется при написании определений функций. Например:

```
$function$
BEGIN
    RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$function$
```

Здесь последовательность `$q$[\t\r\n\v\\]$q$` представляет в долларах текстовую строку `[\t\r\n\v\\]`, которая будет обработана, когда Postgres Pro будет выполнять эту функцию. Но так как

эта последовательность не соответствует внешнему тегу в долларах (`$function$`), с точки зрения внешней строки это просто обычные символы внутри константы.

Тег строки в долларах, если он присутствует, должен соответствовать правилам, определённым для идентификаторов без кавычек, и к тому же не должен содержать знак доллара. Теги регистрозависимы, так что `$tag$String content$tag$` — правильная строка, а `$TAG$String content$tag$` — нет.

Строка в долларах, следующая за ключевым словом или идентификатором, должна отделяться от него пробельными символами, иначе доллар будет считаться продолжением предыдущего идентификатора.

Заключение строк в доллары не является частью стандарта SQL, но часто это более удобный способ записывать сложные строки, чем стандартный вариант с апострофами. Он особенно полезен, когда нужно представить строковую константу внутри другой строки, что часто требуется в определениях процедурных функций. Ограничившись только апострофами, каждую обратную косую черту в приведённом примере пришлось бы записывать четырьмя такими символами, которые бы затем уменьшились до двух при разборе внешней строки, и наконец до одного при обработке внутренней строки во время выполнения функции.

4.1.2.5. Битовые строковые константы

Битовые строковые константы похожи на обычные с дополнительной буквой `B` (заглавной или строчной), добавленной непосредственно перед открывающим апострофом (без промежуточных пробелов), например: `B'1001'`. В битовых строковых константах допускаются лишь символы `0` и `1`.

Битовые константы могут быть записаны и по-другому, в шестнадцатеричном виде, с начальной буквой `X` (заглавной или строчной), например: `X'1FF'`. Такая запись эквивалентна двоичной, только четыре двоичных цифры заменяются одной шестнадцатеричной.

Обе формы записи допускают перенос строк так же, как и обычные строковые константы. Однако заключать в доллары битовые строки нельзя.

4.1.2.6. Числовые константы

Числовые константы могут быть заданы в следующем общем виде:

цифры
цифры.[*цифры*][e[+-]*цифры*]
[*цифры*].*цифры*[e[+-]*цифры*]
*цифры*e[+-]*цифры*

где *цифры* — это одна или несколько десятичных цифр (0..9). До или после десятичной точки (при её наличии) должна быть минимум одна цифра. Как минимум одна цифра должна следовать за обозначением экспоненты (`e`), если оно присутствует. В числовой константе не может быть пробелов или других символов. Заметьте, что любой знак минус или плюс в начале строки не считается частью числа; это оператор, применённый к константе.

Несколько примеров допустимых числовых констант:

42
3.5
4.
.001
5e2
1.925e-3

Числовая константа, не содержащая точки и экспоненты, изначально рассматривается как константа типа `integer`, если её значение умещается в 32-битный тип `integer`; затем как константа типа `bigint`, если её значение умещается в 64-битный `bigint`; в противном случае она принимает тип `numeric`. Константы, содержащие десятичные точки и/или экспоненты, всегда считаются константами типа `numeric`.

Изначально назначенный тип данных числовой константы это только отправная точка для алгоритмов определения типа. В большинстве случаев константа будет автоматически приведена к наиболее подходящему типу для данного контекста. При необходимости вы можете принудительно интерпретировать числовое значение как значение определённого типа, приведя его тип к нужному. Например, вы можете сделать, чтобы числовое значение рассматривалось как имеющее тип `real` (`float4`), написав:

```
REAL '1.23'  -- строковый стиль
1.23::REAL   -- стиль Postgres Pro (исторический)
```

На самом деле это только частные случаи синтаксиса приведения типов, который будет рассматриваться далее.

4.1.2.7. Константы других типов

Константу *обычного* типа можно ввести одним из следующих способов:

```
type 'string'
'string'::type
CAST ( 'string' AS type )
```

Текст строковой константы передаётся процедуре преобразования ввода для типа, обозначенного здесь *type*. Результатом становится константа указанного типа. Явное приведение типа можно опустить, если нужный тип константы определяется однозначно (например, когда она присваивается непосредственно столбцу таблицы), так как в этом случае приведение происходит автоматически.

Строковую константу можно записать, используя как обычный синтаксис SQL, так и формат с долларами.

Также можно записать приведение типов, используя синтаксис функций:

```
typename ( 'string' )
```

но это работает не для всех имён типов; подробнее об этом написано в Подразделе 4.2.9.

Конструкцию `::`, `CAST()` и синтаксис вызова функции можно также использовать для преобразования типов обычных выражений во время выполнения, как описано в Подразделе 4.2.9. Во избежание синтаксической неопределённости, запись *тип* `'`*строка*`'` можно использовать только для указания типа простой текстовой константы. Ещё одно ограничение записи *тип* `'`*строка*`'`: она не работает для массивов; для таких констант следует использовать `::` или `CAST()`.

Синтаксис `CAST()` соответствует SQL, а запись *type* `'`*string*`'` является обобщением стандарта: в SQL такой синтаксис поддерживает только некоторые типы данных, но Postgres Pro позволяет использовать его для всех. Синтаксис с `::` имеет исторические корни в Postgres Pro, как и запись в виде вызова функции.

4.1.3. Операторы

Имя оператора образует последовательность не более чем `NAMEDATALEN`-1 (по умолчанию 63) символов из следующего списка:

+ - * / < > = ~ ! @ # % ^ & | ` ?

Однако для имён операторов есть ещё несколько ограничений:

- Сочетания символов `--` и `/*` не могут присутствовать в имени оператора, так как они будут обозначать начало комментария.
- Многосимвольное имя оператора не может заканчиваться знаком `+` или `-`, если только оно не содержит также один из этих символов:

  ~ ! @ # % ^ & | ` ?

  Например, `@-` — допустимое имя оператора, а `*-` — нет. Благодаря этому ограничению, Postgres Pro может разбирать корректные SQL-запросы без пробелов между компонентами.

Записывая нестандартные SQL-операторы, обычно нужно отделять имена соседних операторов пробелами для однозначности. Например, если вы определили левый унарный оператор с именем `@`, вы не можете написать `X*@Y`, а должны написать `X* @Y`, чтобы Postgres Pro однозначно прочитал это как два оператора, а не один.

4.1.4. Специальные знаки

Некоторые не алфавитно-цифровые символы имеют специальное значение, но при этом не являются операторами. Подробнее их использование будет рассмотрено при описании соответствующего элемента синтаксиса. Здесь они упоминаются только для сведения и обобщения их предназначения.

- Знак доллара (`$`), предваряющий число, используется для представления позиционного параметра в теле определения функции или подготовленного оператора. В других контекстах знак доллара может быть частью идентификатора или строковой константы, заключённой в доллары.
- Круглые скобки (`()`) имеют обычное значение и применяются для группировки выражений и повышения приоритета операций. В некоторых случаях скобки — это необходимая часть синтаксиса определённых SQL-команд.
- Квадратные скобки (`[]`) применяются для выделения элементов массива. Подробнее массивы рассматриваются в Разделе 8.15.
- Запятые (`,`) используются в некоторых синтаксических конструкциях для разделения элементов списка.
- Точка с запятой (`;`) завершает команду SQL. Она не может находиться нигде внутри команды, за исключением строковых констант или идентификаторов в кавычках.
- Двоеточие (`:`) применяется для выборки «срезов» массивов (см. Раздел 8.15.) В некоторых диалектах SQL (например, в Embedded SQL) двоеточие может быть префиксом в имени переменной.
- Звёздочка (`*`) используется в некоторых контекстах как обозначение всех полей строки или составного значения. Она также имеет специальное значение, когда используется как аргумент некоторых агрегатных функций, а именно функций, которым не нужны явные параметры.
- Точка (`.`) используется в числовых константах, а также для отделения имён схемы, таблицы и столбца.

4.1.5. Комментарии

Комментарий — это последовательность символов, которая начинается с двух минусов и продолжается до конца строки, например:

```
-- Это стандартный комментарий SQL
```

Кроме этого, блочные комментарии можно записывать в стиле C:

```
/* многострочный комментарий
 * с вложенностью: /* вложенный блок комментария */
 */
```

где комментарий начинается с `/*` и продолжается до соответствующего вхождения `*/`. Блочные комментарии можно вкладывать друг в друга, как разрешено по стандарту SQL (но не разрешено в C), так что вы можете комментировать большие блоки кода, которые при этом уже могут содержать блоки комментариев.

Комментарий удаляется из входного потока в начале синтаксического анализа и фактически заменяется пробелом.

4.1.6. Приоритеты операторов

В Таблице 4.2 показаны приоритеты и очерёдность операторов, действующие в Postgres Pro. Большинство операторов имеют одинаковый приоритет и вычисляются слева направо. Приоритет и очерёдность операторов жёстко фиксированы в синтаксическом анализаторе.

Иногда вам потребуется добавлять скобки, когда вы комбинируете унарные и бинарные операторы. Например, выражение:

```
SELECT 5 ! - 6;
```

будет разобрано как:

```
SELECT 5 ! (- 6);
```

так как анализатор до последнего не знает, что оператор `!` определён как постфиксный, а не инфиксный (внутренний). Чтобы получить желаемый результат в этом случае, нужно написать:

```
SELECT (5 !) - 6;
```

Такова цена расширяемости.

Таблица 4.2. Приоритет операторов (от большего к меньшему)

Оператор/элемент	Очерёдность	Описание
`.`	слева-направо	разделитель имён таблицы и столбца
`::`	слева-направо	приведение типов в стиле Postgres Pro
`[ ]`	слева-направо	выбор элемента массива
`+ -`	справа-налево	унарный плюс, унарный минус
`^`	слева-направо	возведение в степень
`* / %`	слева-направо	умножение, деление, остаток от деления
`+ -`	слева-направо	сложение, вычитание
(любой другой оператор)	слева-направо	все другие встроенные и пользовательские операторы
`BETWEEN IN LIKE ILIKE SIMILAR`		проверка диапазона, проверка членства, сравнение строк
`< > = <= >= <>`		операторы сравнения
`IS ISNULL NOTNULL`		`IS TRUE`, `IS FALSE`, `IS NULL`, `IS DISTINCT FROM` и т. д.
`NOT`	справа-налево	логическое отрицание
`AND`	слева-направо	логическая конъюнкция
`OR`	слева-направо	логическая дизъюнкция

Заметьте, что правила приоритета операторов также применяются к операторам, определённым пользователем с теми же именами, что и вышеперечисленные встроенные операторы. Например, если вы определите оператор «+» для некоторого нестандартного типа данных, он будет иметь тот же приоритет, что и встроенный оператор «+», независимо от того, что он у вас делает.

Когда в конструкции `OPERATOR` используется имя оператора со схемой, например так:

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

тогда `OPERATOR` имеет приоритет по умолчанию, соответствующий в Таблице 4.2 строке «любой другой оператор». Это не зависит от того, какие именно операторы находятся в конструкции `OPERATOR()`.

Примечание

В PostgreSQL до версии 9.5 действовали немного другие правила приоритета операторов. В частности, операторы <=, >= и <> обрабатывались по общему правилу; проверки IS имели более высокий приоритет; а NOT BETWEEN и связанные конструкции работали несогласованно — в некоторых случаях приоритетнее оказывался оператор NOT, а не BETWEEN. Эти правила были изменены для лучшего соответствия стандарту SQL и для уменьшения путаницы из-за несогласованной обработки логически равнозначных конструкций. В большинстве случаев эти изменения никак не проявятся, либо могут привести к ошибкам типа «нет такого оператора», которые можно разрешить, добавив скобки. Однако возможны особые случаи, когда запрос будет разобран без ошибки, но его поведение может измениться. Если вас беспокоит, не нарушают ли эти изменения незаметно работу вашего приложения, вы можете проверить это, включив конфигурационный параметр operator_precedence_warning и пронаблюдав, не появятся ли предупреждения в журнале.

4.2. Выражения значения

Выражения значения применяются в самых разных контекстах, например в списке результатов команды SELECT, в значениях столбцов в INSERT или UPDATE или в условиях поиска во многих командах. Результат такого выражения иногда называют *скаляром*, чтобы отличить его от результата табличного выражения (который представляет собой таблицу). А сами выражения значения часто называют *скалярными* (или просто *выражениями*). Синтаксис таких выражений позволяет вычислять значения из примитивных частей, используя арифметические, логические и другие операции.

Выражениями значения являются:

- Константа или непосредственное значение
- Ссылка на столбец
- Ссылка на позиционный параметр в теле определения функции или подготовленного оператора
- Выражение с индексом
- Выражение выбора поля
- Применение оператора
- Вызов функции
- Агрегатное выражение
- Вызов оконной функции
- Приведение типов
- Применение правил сортировки
- Скалярный подзапрос
- Конструктор массива
- Конструктор табличной строки
- Кроме того, выражением значения являются скобки (предназначенные для группировки подвыражений и переопределения приоритета)

В дополнение к этому списку есть ещё несколько конструкций, которые можно классифицировать как выражения, хотя они не соответствуют общим синтаксическим правилам. Они обычно имеют вид функции или оператора и будут рассмотрены в соответствующем разделе Главы 9. Пример такой конструкции — предложение IS NULL.

Мы уже обсудили константы в Подразделе 4.1.2. В следующих разделах рассматриваются остальные варианты.

4.2.1. Ссылки на столбцы

Ссылку на столбец можно записать в форме:

отношение.*имя_столбца*

Здесь *отношение* — имя таблицы (возможно, полное, с именем схемы) или её псевдоним, определённый в предложении FROM. Это имя и разделяющую точку можно опустить, если имя столбца уникально среди всех таблиц, задействованных в текущем запросе. (См. также Главу 7.)

4.2.2. Позиционные параметры

Ссылка на позиционный параметр применяется для обращения к значению, переданному в SQL-оператор извне. Параметры используются в определениях SQL-функций и подготовленных операторов. Некоторые клиентские библиотеки также поддерживают передачу значений данных отдельно от самой SQL-команды, и в этом случае параметры позволяют ссылаться на такие значения. Ссылка на параметр записывается в следующей форме:

$*число*

Например, рассмотрим следующее определение функции dept:

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE name = $1 $$
    LANGUAGE SQL;
```

Здесь $1 всегда будет ссылаться на значение первого аргумента функции.

4.2.3. Индексы элементов

Если в выражении вы имеете дело с массивом, то можно извлечь определённый его элемент, написав:

выражение[*индекс*]

или несколько соседних элементов («срез массива»):

выражение[*нижний_индекс*:*верхний_индекс*]

(Здесь квадратные скобки [ ] должны присутствовать буквально.) Каждый *индекс* сам по себе является выражением, результат которого округляется к ближайшему целому.

В общем случае *выражение* массива должно заключаться в круглые скобки, но их можно опустить, когда выражение с индексом — это просто ссылка на столбец или позиционный параметр. Кроме того, можно соединить несколько индексов, если исходный массив многомерный. Например:

```
моя_таблица.столбец_массив[4]
моя_таблица.столбец_массив_2d[17][34]
$1[10:42]
(функция_массив(a,b))[42]
```

В последней строке круглые скобки необходимы. Подробнее массивы рассматриваются в Разделе 8.15.

4.2.4. Выбор поля

Если результат выражения — значение составного типа (строка таблицы), тогда определённое поле этой строки можно извлечь, написав:

выражение.*имя_поля*

В общем случае *выражение* такого типа должно заключаться в круглые скобки, но их можно опустить, когда это ссылка на таблицу или позиционный параметр. Например:

```
моя_таблица.столбец
$1.столбец
(функция_кортеж(a,b)).стол3
```

(Таким образом, полная ссылка на столбец — это просто частный случай выбора поля.) Важный особый случай здесь — извлечение поля из столбца составного типа:

```
(составной_столбец).поле
(моя_таблица.составной_столбец).поле
```

Здесь скобки нужны, чтобы показать, что `составной_столбец` — это имя столбца, а не таблицы, и что `моя_таблица` — имя таблицы, а не схемы.

Вы можете запросить все поля составного значения, написав `.*`:

```
(составной_столбец).*
```

Эта запись действует по-разному в зависимости от контекста; подробнее об этом говорится в Подразделе 8.16.5.

4.2.5. Применение оператора

Существуют три возможных синтаксиса применения операторов:

выражение оператор выражение (бинарный инфиксный оператор)
оператор выражение (унарный префиксный оператор)
выражение оператор (унарный постфиксный оператор)

где *оператор* соответствует синтаксическим правилам, описанным в Подразделе 4.1.3, либо это одно из ключевых слов `AND`, `OR` и `NOT`, либо полное имя оператора в форме:

```
OPERATOR(схема.имя_оператора)
```

Существование конкретных операторов и их тип (унарный или бинарный) зависит от того, как и какие операторы определены системой и пользователем. Встроенные операторы описаны в Главе 9.

4.2.6. Вызовы функций

Вызов функции записывается просто как имя функции (возможно, дополненное именем схемы) и список аргументов в скобках:

```
имя_функции ([выражение [, выражение ... ]])
```

Например, так вычисляется квадратный корень из 2:

```
sqrt(2)
```

Список встроенных функций приведён в Главе 9. Пользователь также может определить и другие функции.

Выполняя запросы в базе данных, где одни пользователи могут не доверять другим, в записи вызовов функций соблюдайте меры предосторожности, описанные в Разделе 10.3.

Аргументам могут быть присвоены необязательные имена. Подробнее об этом см. Раздел 4.3.

Примечание

Функцию, принимающую один аргумент составного типа, можно также вызывать, используя синтаксис выбора поля, и наоборот, выбор поля можно записать в функциональном стиле. То есть записи `col(table)` и `table.col` равносильны и взаимозаменяемы. Это поведение не оговорено стандартом SQL, но реализовано в Postgres Pro, так как это позволяет использовать функции для эмуляции «вычисляемых полей». Подробнее это описано в Подразделе 8.16.5.

4.2.7. Агрегатные выражения

Агрегатное выражение представляет собой применение агрегатной функции к строкам, выбранным запросом. Агрегатная функция сводит множество входных значений к одному выходному, как например, сумма или среднее. Агрегатное выражение может записываться следующим образом:

```
агрегатная_функция (выражение [ , ... ] [ предложение_order_by ] ) [ FILTER
 ( WHERE условие_фильтра ) ]
агрегатная_функция (ALL выражение [ , ... ] [ предложение_order_by ] ) [ FILTER
 ( WHERE условие_фильтра ) ]
агрегатная_функция (DISTINCT выражение [ , ... ] [ предложение_order_by ] ) [ FILTER
 ( WHERE условие_фильтра ) ]
агрегатная_функция ( * ) [ FILTER ( WHERE условие_фильтра ) ]
агрегатная_функция ( [ выражение [ , ... ] ] ) WITHIN GROUP ( предложение_order_by )
 [ FILTER ( WHERE условие_фильтра ) ]
```

Здесь *агрегатная_функция* — имя ранее определённой агрегатной функции (возможно, дополненное именем схемы), *выражение* — любое выражение значения, не содержащее в себе агрегатного выражения или вызова оконной функции. Необязательные предложения *предложение_order_by* и *условие_фильтра* описываются ниже.

В первой форме агрегатного выражения агрегатная функция вызывается для каждой строки. Вторая форма эквивалентна первой, так как указание `ALL` подразумевается по умолчанию. В третьей форме агрегатная функция вызывается для всех различных значений выражения (или набора различных значений, для нескольких выражений), выделенных во входных данных. В четвёртой форме агрегатная функция вызывается для каждой строки, так как никакого конкретного значения не указано (обычно это имеет смысл только для функции `count(*)`). В последней форме используются *сортирующие* агрегатные функции, которые будут описаны ниже.

Большинство агрегатных функций игнорируют значения NULL, так что строки, для которых выражения выдают одно или несколько значений NULL, отбрасываются. Это можно считать истинным для всех встроенных операторов, если явно не говорится об обратном.

Например, `count(*)` подсчитает общее количество строк, а `count(f1)` только количество строк, в которых `f1` не NULL (так как `count` игнорирует NULL), а `count(distinct f1)` подсчитает число различных и отличных от NULL значений столбца `f1`.

Обычно строки данных передаются агрегатной функции в неопределённом порядке и во многих случаях это не имеет значения, например функция `min` выдаёт один и тот же результат независимо от порядка поступающих данных. Однако некоторые агрегатные функции (такие как `array_agg` и `string_agg`) выдают результаты, зависящие от порядка данных. Для таких агрегатных функций можно добавить *предложение_order_by* и задать нужный порядок. Это *предложение_order_by* имеет тот же синтаксис, что и предложение `ORDER BY` на уровне запроса, как описано в Разделе 7.5, за исключением того, что его выражения должны быть просто выражениями, а не именами результирующих столбцов или числами. Например:

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

Заметьте, что при использовании агрегатных функций с несколькими аргументами, предложение `ORDER BY` идёт после всех аргументов. Например, надо писать так:

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

а не так:

```
SELECT string_agg(a ORDER BY a, ',') FROM table;  -- неправильно
```

Последний вариант синтаксически допустим, но он представляет собой вызов агрегатной функции одного аргумента с двумя ключами `ORDER BY` (при этом второй не имеет смысла, так как это константа).

Если *предложение_order_by* дополнено указанием DISTINCT, тогда все выражения ORDER BY должны соответствовать обычным аргументам агрегатной функции; то есть вы не можете сортировать строки по выражению, не включённому в список DISTINCT.

> **Примечание**
>
> Возможность указывать и DISTINCT, и ORDER BY в агрегатной функции — это расширение Postgres Pro.

При добавлении ORDER BY в обычный список аргументов агрегатной функции, описанном до этого, выполняется сортировка входных строк для универсальных и статистических агрегатных функций, для которых сортировка необязательна. Но есть подмножество агрегатных функций, *сортирующие агрегатные функции*, для которых *предложение_order* является *обязательным*, обычно потому, что вычисление этой функции имеет смысл только при определённой сортировке входных строк. Типичными примерами сортирующих агрегатных функций являются вычисления ранга и процентиля. Для сортирующей агрегатной функции *предложение_order_by* записывается внутри WITHIN GROUP (...), что иллюстрирует последний пример, приведённый выше. Выражения в *предложении_order_by* вычисляются однократно для каждой входной строки как аргументы обычной агрегатной функции, сортируются в соответствии с требованием *предложения_order_by* и поступают в агрегатную функции как входящие аргументы. (Если же *предложение_order_by* находится не в WITHIN GROUP, оно не передаётся как аргумент(ы) агрегатной функции.) Выражения-аргументы, предшествующие WITHIN GROUP, (если они есть), называются *непосредственными аргументами*, а выражения, указанные в *предложении_order_by* — *агрегируемыми аргументами*. В отличие от аргументов обычной агрегатной функции, непосредственные аргументы вычисляются однократно для каждого вызова функции, а не для каждой строки. Это значит, что они могут содержать переменные, только если эти переменные сгруппированы в GROUP BY; это суть то же ограничение, что действовало бы, будь эти непосредственные аргументы вне агрегатного выражения. Непосредственные аргументы обычно используются, например, для указания значения процентиля, которое имеет смысл, только если это конкретное число для всего расчёта агрегатной функции. Список непосредственных аргументов может быть пуст; в этом случае запишите просто (), но не (*). (На самом деле Postgres Pro примет обе записи, но только первая соответствует стандарту SQL.)

Пример вызова сортирующей агрегатной функции:

```
SELECT percentile_cont(0.5) WITHIN GROUP (ORDER BY income) FROM households;
 percentile_cont
-----------------
           50489
```

она получает 50-ый процентиль, или медиану, значения столбца income из таблицы households. В данном случае 0.5 — это непосредственный аргумент; если бы дробь процентиля менялась от строки к строке, это не имело бы смысла.

Если добавлено предложение FILTER, агрегатной функции подаются только те входные строки, для которых *условие_фильтра* вычисляется как истинное; другие строки отбрасываются. Например:

```
SELECT
    count(*) AS unfiltered,
    count(*) FILTER (WHERE i < 5) AS filtered
FROM generate_series(1,10) AS s(i);
 unfiltered | filtered
------------+----------
         10 |        4
(1 row)
```

Предопределённые агрегатные функции описаны в Разделе 9.20. Пользователь также может определить другие агрегатные функции.

Агрегатное выражение может фигурировать только в списке результатов или в предложении HAVING команды SELECT. Во всех остальных предложениях, например WHERE, они запрещены, так как эти предложения логически вычисляются до того, как формируются результаты агрегатных функций.

Когда агрегатное выражение используется в подзапросе (см. Подраздел 4.2.11 и Раздел 9.22), оно обычно вычисляется для всех строк подзапроса. Но если в аргументах (или в *условии_filter*) агрегатной функции есть только переменные внешнего уровня, агрегатная функция относится к ближайшему внешнему уровню и вычисляется для всех строк соответствующего запроса. Такое агрегатное выражение в целом является внешней ссылкой для своего подзапроса и на каждом вычислении считается константой. При этом допустимое положение агрегатной функции ограничивается списком результатов и предложением HAVING на том уровне запросов, где она находится.

4.2.8. Вызовы оконных функций

Вызов оконной функции представляет собой применение функции, подобной агрегатной, к некоторому набору строк, выбранному запросом. В отличие от вызовов не оконных агрегатных функций, они не связаны с группировкой выбранных строк в одну — каждая строка остаётся отдельной в результате запроса. Однако оконная функция имеет доступ ко всем строкам, вошедшим в группу текущей строки согласно указанию группировки (списку PARTITION BY) в вызове оконной функции. Вызов оконной функции может иметь следующие формы:

```
имя_функции ([выражение [, выражение ... ]]) [ FILTER ( WHERE предложение_фильтра ) ]
 OVER имя_окна
имя_функции ([выражение [, выражение ... ]]) [ FILTER ( WHERE предложение_фильтра ) ]
 OVER ( определение_окна )
имя_функции ( * ) [ FILTER ( WHERE предложение_фильтра ) ] OVER имя_окна
имя_функции ( * ) [ FILTER ( WHERE предложение_фильтра ) ] OVER ( определение_окна )
```

Здесь *определение_окна* записывается в виде

```
[ имя_существующего_окна ]
[ PARTITION BY выражение [, ...] ]
[ ORDER BY выражение [ ASC | DESC | USING оператор ] [ NULLS { FIRST | LAST } ]
 [, ...] ]
[ определение_рамки ]
```

и необязательное *определение_рамки* может иметь вид:

```
{ RANGE | ROWS } начало_рамки
{ RANGE | ROWS } BETWEEN начало_рамки AND конец_рамки
```

Здесь *начало_рамки* и *конец_рамки* задаются одним из следующих способов:

```
UNBOUNDED PRECEDING
значение PRECEDING
CURRENT ROW
значение FOLLOWING
UNBOUNDED FOLLOWING
```

Здесь *выражение* — это любое выражение значения, не содержащее вызовов оконных функций.

имя_окна — ссылка на именованное окно, определённое предложением WINDOW в данном запросе. Также возможно написать в скобках полное *определение_окна*, используя тот же синтаксис определения именованного окна в предложении WINDOW; подробнее это описано в справке по SELECT. Стоит отметить, что запись OVER имя_окна не полностью равнозначна OVER (имя_окна ...); последний вариант подразумевает копирование и изменение определения окна и не будет допустимым, если определение этого окна включает определение рамки.

Указание PARTITION BY группирует строки запроса в *разделы*, которые затем обрабатываются оконной функцией независимо друг от друга. PARTITION BY работает подобно предложению GROUP

BY на уровне запроса, за исключением того, что его аргументы всегда просто выражения, а не имена выходных столбцов или числа. Без PARTITION BY все строки, выдаваемые запросом, рассматриваются как один раздел. Указание ORDER BY определяет порядок, в котором оконная функция обрабатывает строки раздела. Оно так же подобно предложению ORDER BY на уровне запроса и так же не принимает имена выходных столбцов или числа. Без ORDER BY строки обрабатываются в неопределённом порядке.

определение_рамки задаёт набор строк, образующих *рамку окна*, которая представляет собой подмножество строк текущего раздела и используется для оконных функций, работающих с рамкой, а не со всем разделом. Рамку можно указать в режимах RANGE или ROWS; в любом случае она начинается с положения *начало_рамки* и заканчивается положением *конец_рамки*. Если *конец_рамки* опущен, подразумевается CURRENT ROW (текущая строка).

Если *начало_рамки* задано как UNBOUNDED PRECEDING, рамка начинается с первой строки раздела, а если *конец_рамки* определён как UNBOUNDED FOLLOWING, рамка заканчивается последней строкой раздела.

В режиме RANGE *начало_рамки*, заданное как CURRENT ROW, определяет в качестве начала первую *родственную* строку (строку, которую ORDER BY считает равной текущей), тогда как *конец_рамки*, заданный как CURRENT ROW, определяет концом рамки последнюю родственную (для ORDER BY) строку. В режиме ROWS вариант CURRENT ROW просто обозначает текущую строку.

Варианты *значение* PRECEDING и *значение* FOLLOWING допускаются только в режиме ROWS. Они указывают, что рамка начинается или заканчивается со сдвигом на заданное число строк перед или после заданной строки. Здесь *значение* должно быть целочисленным выражением, не содержащим переменные, агрегатные или оконные функции, и может быть нулевым, что будет означать выбор текущей строки.

По умолчанию рамка определяется как RANGE UNBOUNDED PRECEDING, что равносильно расширенному определению RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. С указанием ORDER BY это означает, что рамка будет включать все строки от начала раздела до последней строки, родственной текущей (для ORDER BY). Без ORDER BY в рамку включаются все строки раздела, так как все они считаются родственными текущей.

Действуют также ограничения: *начало_рамки* не может определяться как UNBOUNDED FOLLOWING, а *конец_рамки* — UNBOUNDED PRECEDING, и *конец_рамки* не может определяться раньше, чем *начало_рамки* — например, запись RANGE BETWEEN CURRENT ROW AND *значение* PRECEDING недопустима.

Если добавлено предложение FILTER, оконной функции подаются только те входные строки, для которых *условие_фильтра* вычисляется как истинное; другие строки отбрасываются. Предложение FILTER допускается только для агрегирующих оконных функций.

Встроенные оконные функции описаны в Таблице 9.57, но пользователь может расширить этот набор, создавая собственные функции. Кроме того, в качестве оконных функций можно использовать любые встроенные или пользовательские универсальные, а также статистические агрегатные функции. (Сортирующие и гипотезирующие агрегатные функции в настоящее время использовать в качестве оконных нельзя.)

Запись со звёздочкой (*) применяется при вызове не имеющих параметров агрегатных функций в качестве оконных, например `count(*) OVER (PARTITION BY x ORDER BY y)`. Звёздочка (*) обычно не применяется для исключительно оконных функций. Такие функции не допускают использования DISTINCT и ORDER BY в списке аргументов функции.

Вызовы оконных функций разрешены в запросах только в списке SELECT и в предложении ORDER BY.

Дополнительно об оконных функциях можно узнать в Разделе 3.5, Разделе 9.21 и Подразделе 7.2.5.

4.2.9. Приведения типов

Приведение типа определяет преобразование данных из одного типа в другой. Postgres Pro воспринимает две равносильные записи приведения типов:

```
CAST ( выражение AS тип )
выражение::тип
```

Запись с `CAST` соответствует стандарту SQL, тогда как вариант с `::` — историческое наследие Postgres Pro.

Когда приведению подвергается значение выражения известного типа, происходит преобразование типа во время выполнения. Это приведение будет успешным, только если определён подходящий оператор преобразования типов. Обратите внимание на небольшое отличие от приведения констант, описанного в Подразделе 4.1.2.7. Приведение строки в чистом виде представляет собой начальное присваивание строковой константы и оно будет успешным для любого типа (конечно, если строка содержит значение, приемлемое для данного типа данных).

Явное приведение типа можно опустить, если возможно однозначно определить, какой тип должно иметь выражение (например, когда оно присваивается столбцу таблицы); в таких случаях система автоматически преобразует тип. Однако автоматическое преобразование выполняется только для приведений с пометкой «допускается неявное применение» в системных каталогах. Все остальные приведения должны записываться явно. Это ограничение позволяет избежать сюрпризов с неявным преобразованием.

Также можно записать приведение типа как вызов функции:

```
имя_типа ( выражение )
```

Однако это будет работать только для типов, имена которых являются также допустимыми именами функций. Например, `double precision` так использовать нельзя, а `float8` (альтернативное название того же типа) — можно. Кроме того, имена типов `interval`, `time` и `timestamp` из-за синтаксического конфликта можно использовать в такой записи только в кавычках. Таким образом, запись приведения типа в виде вызова функции провоцирует несоответствия и, возможно, лучше будет её не применять.

> **Примечание**
>
> Приведение типа, представленное в виде вызова функции, на самом деле соответствует внутреннему механизму. Даже при использовании двух стандартных типов записи внутри происходит вызов зарегистрированной функции, выполняющей преобразование. По соглашению именем такой функции преобразования является имя выходного типа, и таким образом запись «в виде вызова функции» есть не что иное, как прямой вызов нижележащей функции преобразования. При создании переносимого приложения на это поведение, конечно, не следует рассчитывать. Подробнее это описано в справке CREATE CAST.

4.2.10. Применение правил сортировки

Предложение `COLLATE` переопределяет правило сортировки выражения. Оно добавляется после выражения:

```
выражение COLLATE правило_сортировки
```

где *правило_сортировки* — идентификатор правила, возможно дополненный именем схемы. Предложение `COLLATE` связывает выражение сильнее, чем операторы, так что при необходимости следует использовать скобки.

Если правило сортировки не определено явно, система либо выбирает его по столбцам, которые используются в выражении, либо, если таких столбцов нет, переключается на установленное для базы данных правило сортировки по умолчанию.

Предложение COLLATE имеет два распространённых применения: переопределение порядка сортировки в предложении ORDER BY, например:

```
SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE "C";
```

и переопределение правил сортировки при вызове функций или операторов, возвращающих языкозависимые результаты, например:

```
SELECT * FROM tbl WHERE a > 'foo' COLLATE "C";
```

Заметьте, что в последнем случае предложение COLLATE добавлено к аргументу оператора, на действие которого мы хотим повлиять. При этом не имеет значения, к какому именно аргументу оператора или функции добавляется COLLATE, так как правило сортировки, применяемое к оператору или функции, выбирается при рассмотрении всех аргументов, а явное предложение COLLATE переопределяет правила сортировки для всех других аргументов. (Однако добавление разных предложений COLLATE к нескольким аргументам будет ошибкой. Подробнее об этом см. Раздел 22.2.) Таким образом, эта команда выдаст тот же результат:

```
SELECT * FROM tbl WHERE a COLLATE "C" > 'foo';
```

Но это будет ошибкой:

```
SELECT * FROM tbl WHERE (a > 'foo') COLLATE "C";
```

здесь правило сортировки нельзя применить к результату оператора >, который имеет несравниваемый тип данных boolean.

4.2.11. Скалярные подзапросы

Скалярный подзапрос — это обычный запрос SELECT в скобках, который возвращает ровно одну строку и один столбец. (Написание запросов освещается в Главе 7.) После выполнения запроса SELECT его единственный результат используется в окружающем его выражении. В качестве скалярного подзапроса нельзя использовать запросы, возвращающие более одной строки или столбца. (Но если в результате выполнения подзапрос не вернёт строк, скалярный результат считается равным NULL.) В подзапросе можно ссылаться на переменные из окружающего запроса; в процессе одного вычисления подзапроса они будут считаться константами. Другие выражения с подзапросами описаны в Разделе 9.22.

Например, следующий запрос находит самый населённый город в каждом штате:

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)
    FROM states;
```

4.2.12. Конструкторы массивов

Конструктор массива — это выражение, которое создаёт массив, определяя значения его элементов. Конструктор простого массива состоит из ключевого слова ARRAY, открывающей квадратной скобки [, списка выражений (разделённых запятыми), задающих значения элементов массива, и закрывающей квадратной скобки]. Например:

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
(1 row)
```

По умолчанию типом элементов массива считается общий тип для всех выражений, определённый по правилам, действующим и для конструкций UNION и CASE (см. Раздел 10.5). Вы можете переопределить его явно, приведя конструктор массива к требуемому типу, например:

```
SELECT ARRAY[1,2,22.7]::integer[];
  array
----------
 {1,2,23}
(1 row)
```

Это равносильно тому, что привести к нужному типу каждое выражение по отдельности. Подробнее приведение типов описано в Подразделе 4.2.9.

Многомерные массивы можно образовывать, вкладывая конструкторы массивов. При этом во внутренних конструкторах слово ARRAY можно опускать. Например, результат работы этих конструкторов одинаков:

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)

SELECT ARRAY[[1,2],[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)
```

Многомерные массивы должны быть прямоугольными, и поэтому внутренние конструкторы одного уровня должны создавать вложенные массивы одинаковой размерности. Любое приведение типа, применённое к внешнему конструктору ARRAY, автоматически распространяется на все внутренние.

Элементы многомерного массива можно создавать не только вложенными конструкторами ARRAY, но и другими способами, позволяющими получить массивы нужного типа. Например:

```
CREATE TABLE arr(f1 int[], f2 int[]);

INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
                     array
------------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
(1 row)
```

Вы можете создать и пустой массив, но так как массив не может быть не типизированным, вы должны явно привести пустой массив к нужному типу. Например:

```
SELECT ARRAY[]::integer[];
 array
-------
 {}
(1 row)
```

Также возможно создать массив из результатов подзапроса. В этом случае конструктор массива записывается так же с ключевым словом ARRAY, за которым в круглых скобках следует подзапрос. Например:

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
                                  array
------------------------------------------------------------------------
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31,2412,2413}
(1 row)

SELECT ARRAY(SELECT ARRAY[i, i*2] FROM generate_series(1,5) AS a(i));
              array
----------------------------------
 {{1,2},{2,4},{3,6},{4,8},{5,10}}
(1 row)
```

Такой подзапрос должен возвращать один столбец. Если этот столбец имеет тип, отличный от массива, результирующий одномерный массив будет включать элементы для каждой строки-результата подзапроса и типом элемента будет тип столбца результата. Если же тип столбца — массив, будет создан массив того же типа, но большей размерности; в любом случае во всех строках подзапроса должны выдаваться массивы одинаковой размерности, чтобы можно было получить прямоугольный результат.

Индексы массива, созданного конструктором `ARRAY`, всегда начинаются с одного. Подробнее о массивах вы узнаете в Разделе 8.15.

4.2.13. Конструкторы табличных строк

Конструктор табличной строки — это выражение, создающее строку или кортеж (или составное значение) из значений его аргументов-полей. Конструктор строки состоит из ключевого слова `ROW`, открывающей круглой скобки, нуля или нескольких выражений (разделённых запятыми), определяющих значения полей, и закрывающей скобки. Например:

```
SELECT ROW(1,2.5,'this is a test');
```

Если в списке более одного выражения, ключевое слово `ROW` можно опустить.

Конструктор строки поддерживает запись *составное_значение*`.*`, при этом данное значение будет развёрнуто в список элементов, так же, как в записи `.*` на верхнем уровне списка `SELECT` (см. Подраздел 8.16.5). Например, если таблица `t` содержит столбцы `f1` и `f2`, эти записи равнозначны:

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

> **Примечание**
>
> До версии PostgreSQL 8.2 запись `.*` не разворачивалась в конструкторах строк, так что выражение `ROW(t.*, 42)` создавало составное значение из двух полей, в котором первое поле так же было составным. Новое поведение обычно более полезно. Если вам нужно получить прежнее поведение, чтобы одно значение строки было вложено в другое, напишите внутреннее значение без `.*`, например: `ROW(t, 42)`.

По умолчанию значение, созданное выражением `ROW`, имеет тип анонимной записи. Если необходимо, его можно привести к именованному составному типу — либо к типу строки таблицы, либо составному типу, созданному оператором `CREATE TYPE AS`. Явное приведение может потребоваться для достижения однозначности. Например:

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);

CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- Приведение не требуется, так как существует только одна getf1()
SELECT getf1(ROW(1,2.5,'this is a test'));
 getf1
-------
     1
(1 row)

CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- Теперь приведение необходимо для однозначного выбора функции:
SELECT getf1(ROW(1,2.5,'this is a test'));
ОШИБКА:  функция getf1(record) не уникальна
```

```
SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
 getf1
-------
     1
(1 row)

SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS myrowtype));
 getf1
-------
    11
(1 row)
```

Используя конструктор строк (кортежей), можно создавать составное значение для сохранения в столбце составного типа или для передачи функции, принимающей составной параметр. Также вы можете сравнить два составных значения или проверить их с помощью `IS NULL` или `IS NOT NULL`, например:

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');

  -- выбрать все строки, содержащие только NULL
SELECT ROW(table.*) IS NULL FROM table;
```

Подробнее см. Раздел 9.23. Конструкторы строк также могут использоваться в сочетании с подзапросами, как описано в Разделе 9.22.

4.2.14. Правила вычисления выражений

Порядок вычисления подвыражений не определён. В частности, аргументы оператора или функции не обязательно вычисляются слева направо или в любом другом фиксированном порядке.

Более того, если результат выражения можно получить, вычисляя только некоторые его части, тогда другие подвыражения не будут вычисляться вовсе. Например, если написать:

```
SELECT true OR somefunc();
```

тогда функция `somefunc()` не будет вызываться (возможно). То же самое справедливо для записи:

```
SELECT somefunc() OR true;
```

Заметьте, что это отличается от «оптимизации» вычисления логических операторов слева направо, реализованной в некоторых языках программирования.

Как следствие, в сложных выражениях не стоит использовать функции с побочными эффектами. Особенно опасно рассчитывать на порядок вычисления или побочные эффекты в предложениях `WHERE` и `HAVING`, так как эти предложения тщательно оптимизируются при построении плана выполнения. Логические выражения (сочетания `AND`/`OR`/`NOT`) в этих предложениях могут быть видоизменены любым способом, допустимым законами Булевой алгебры.

Когда порядок вычисления важен, его можно зафиксировать с помощью конструкции `CASE` (см. Раздел 9.17). Например, такой способ избежать деления на ноль в предложении `WHERE` ненадёжен:

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

Безопасный вариант:

```
SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;
```

Применяемая так конструкция `CASE` защищает выражение от оптимизации, поэтому использовать её нужно только при необходимости. (В данном случае было бы лучше решить проблему, переписав условие как `y > 1.5*x`.)

Однако `CASE` не всегда спасает в подобных случаях. Показанный выше приём плох тем, что не предотвращает раннее вычисление константных подвыражений. Как описано в Разделе 35.6,

функции и операторы, помеченные как IMMUTABLE, могут вычисляться при планировании, а не выполнении запроса. Поэтому в примере

```
SELECT CASE WHEN x > 0 THEN x ELSE 1/0 END FROM tab;
```

, скорее всего, произойдёт деление на ноль из-за того, что планировщик попытается упростить константное подвыражение, даже если во всех строках в таблице x > 0, а значит во время выполнения ветвь ELSE никогда не будет выполняться.

Хотя этот конкретный пример может показаться надуманным, похожие ситуации, в которых неявно появляются константы, могут возникать и в запросах внутри функций, так как значения аргументов функции и локальных переменных при планировании могут быть заменены константами. Поэтому, например, в функциях PL/pgSQL гораздо безопаснее для защиты от рискованных вычислений использовать конструкцию IF-THEN-ELSE, чем выражение CASE.

Ещё один подобный недостаток этого подхода в том, что CASE не может предотвратить вычисление заключённого в нём агрегатного выражения, так как агрегатные выражения вычисляются перед всеми остальными в списке SELECT или предложении HAVING. Например, в следующем запросе может возникнуть ошибка деления на ноль, несмотря на то, что он вроде бы защищён от неё:

```
SELECT CASE WHEN min(employees) > 0
            THEN avg(expenses / employees)
       END
    FROM departments;
```

Агрегатные функции min() и avg() вычисляются независимо по всем входным строкам, так что если в какой-то строке поле employees окажется равным нулю, деление на ноль произойдёт раньше, чем станет возможным проверить результат функции min(). Поэтому, чтобы проблемные входные строки изначально не попали в агрегатную функцию, следует воспользоваться предложениями WHERE или FILTER.

4.3. Вызов функций

Postgres Pro позволяет вызывать функции с именованными параметрами, используя запись с *позиционной* или *именной* передачей аргументов. Именная передача особенно полезна для функций со множеством параметров, так как она делает связь параметров и аргументов более явной и надёжной. В позиционной записи значения аргументов функции указываются в том же порядке, в каком они описаны в определении функции. При именной передаче аргументы сопоставляются с параметрами функции по именам и указывать их можно в любом порядке. Для каждого варианта вызова также учитывайте влияние типов аргументов функций, описанное в Разделе 10.3.

При записи любым способом параметры, для которых в определении функции заданы значения по умолчанию, можно вовсе не указывать. Но это особенно полезно при именной передаче, так как опустить можно любой набор параметров, тогда как при позиционной параметры можно опускать только последовательно, справа налево.

Postgres Pro также поддерживает *смешанную* передачу, когда параметры передаются и по именам, и по позиции. В этом случае позиционные параметры должны идти перед параметрами, передаваемыми по именам.

Мы рассмотрим все три варианта записи на примере следующей функции:

```
CREATE FUNCTION concat_lower_or_upper(a text, b text,
  uppercase boolean DEFAULT false)
RETURNS text
AS
$$
 SELECT CASE
        WHEN $3 THEN UPPER($1 || ' ' || $2)
        ELSE LOWER($1 || ' ' || $2)
```

```
        END;
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

Функция `concat_lower_or_upper` имеет два обязательных параметра: `a` и `b`. Кроме того, есть один необязательный параметр `uppercase`, который по умолчанию имеет значение `false`. Аргументы `a` и `b` будут сложены вместе и переведены в верхний или нижний регистр, в зависимости от параметра `uppercase`. Остальные тонкости реализации функции сейчас не важны (подробнее о них рассказано в Главе 35).

4.3.1. Позиционная передача

Позиционная передача — это традиционный механизм передачи аргументов функции в Postgres Pro. Пример такой записи:

```
SELECT concat_lower_or_upper('Hello', 'World', true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

Все аргументы указаны в заданном порядке. Результат возвращён в верхнем регистре, так как параметр `uppercase` имеет значение `true`. Ещё один пример:

```
SELECT concat_lower_or_upper('Hello', 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

Здесь параметр `uppercase` опущен, и поэтому он принимает значение по умолчанию (`false`), и результат переводится в нижний регистр. В позиционной записи любые аргументы с определённым значением по умолчанию можно опускать справа налево.

4.3.2. Именная передача

При именной передаче для аргумента добавляется имя, которое отделяется от выражения значения знаками `=>`. Например:

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

Здесь аргумент `uppercase` был так же опущен, так что он неявно получил значение `false`. Преимуществом такой записи является возможность записывать аргументы в любом порядке, например:

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)

SELECT concat_lower_or_upper(a => 'Hello', uppercase => true, b => 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

Для обратной совместимости поддерживается и старый синтаксис с «:=»:

```
SELECT concat_lower_or_upper(a := 'Hello', uppercase := true, b := 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

4.3.3. Смешанная передача

При смешанной передаче параметры передаются и по именам, и по позиции. Однако как уже было сказано, именованные аргументы не могут стоять перед позиционными. Например:

```
SELECT concat_lower_or_upper('Hello', 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

В данном запросе аргументы `a` и `b` передаются по позиции, а `uppercase` — по имени. Единственное обоснование такого вызова здесь — он стал чуть более читаемым. Однако для более сложных функций с множеством аргументов, часть из которых имеют значения по умолчанию, именная или смешанная передача позволяют записать вызов эффективнее и уменьшить вероятность ошибок.

> **Примечание**
>
> Именная и смешанная передача в настоящий момент не может использоваться при вызове агрегатной функции (но они допускаются, если агрегатная функция используется в качестве оконной).

Глава 5. Определение данных

Эта глава рассказывает, как создавать структуры базы данных, в которых будут храниться данные. В реляционной базе данных данные хранятся в таблицах, так что большая часть этой главы будет посвящена созданию и изменению таблиц, а также средствам управления данными в них. Затем мы обсудим, как таблицы можно объединять в схемы и как ограничивать доступ к ним. Наконец, мы кратко рассмотрим другие возможности, связанные с хранением данных, в частности наследование, секционирование таблиц, представления, функции и триггеры.

5.1. Основы таблиц

Таблица в реляционной базе данных похожа на таблицу на бумаге: она так же состоит из строк и столбцов. Число и порядок столбцов фиксированы, а каждый столбец имеет имя. Число строк переменно — оно отражает текущее количество находящихся в ней данных. SQL не даёт никаких гарантий относительно порядка строк таблицы. При чтении таблицы строки выводятся в произвольном порядке, если только явно не требуется сортировка. Подробнее это рассматривается в Главе 7. Более того, SQL не назначает строкам уникальные идентификаторы, так что можно иметь в таблице несколько полностью идентичных строк. Это вытекает из математической модели, которую реализует SQL, но обычно такое дублирование нежелательно. Позже в этой главе мы увидим, как его избежать.

Каждому столбцу сопоставлен тип данных. Тип данных ограничивает набор допустимых значений, которые можно присвоить столбцу, и определяет смысловое значение данных для вычислений. Например, в столбец числового типа нельзя записать обычные текстовые строки, но зато его данные можно использовать в математических вычислениях. И наоборот, если столбец имеет тип текстовой строки, для него допустимы практически любые данные, но он непригоден для математических действий (хотя другие операции, например конкатенация строк, возможны).

В Postgres Pro есть внушительный набор встроенных типов данных, удовлетворяющий большинство приложений. Пользователи также могут определять собственные типы данных. Большинство встроенных типов данных имеют понятные имена и семантику, так что мы отложим их подробное рассмотрение до Главы 8. Наиболее часто применяются следующие типы данных: `integer` для целых чисел, `numeric` для чисел, которые могут быть дробными, `text` для текстовых строк, `date` для дат, `time` для времени и `timestamp` для значений, включающих дату и время.

Для создания таблицы используется команда CREATE TABLE. В этой команде вы должны указать как минимум имя новой таблицы и имена и типы данных каждого столбца. Например:

```
CREATE TABLE my_first_table (
    first_column text,
    second_column integer
);
```

Так вы создадите таблицу `my_first_table` с двумя столбцами. Первый столбец называется `first_column` и имеет тип данных `text`; второй столбец называется `second_column` и имеет тип `integer`. Имена таблицы и столбцов соответствуют синтаксису идентификаторов, описанному в Подразделе 4.1.1. Имена типов также являются идентификаторами, хотя есть некоторые исключения. Заметьте, что список столбцов заключается в скобки, а его элементы разделяются запятыми.

Конечно, предыдущий пример ненатурален. Обычно в именах таблиц и столбцов отражается, какие данные они будут содержать. Поэтому давайте взглянем на более реалистичный пример:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

(Тип `numeric` может хранить дробные числа, в которых обычно выражаются денежные суммы.)

> **Подсказка**
>
> Когда вы создаёте много взаимосвязанных таблиц, имеет смысл заранее выбрать единый шаблон именования таблиц и столбцов. Например, решить, будут ли в именах таблиц использоваться существительные во множественном или в единственном числе (есть соображения в пользу каждого варианта).

Число столбцов в таблице не может быть бесконечным. Это число ограничивается максимумом в пределах от 250 до 1600, в зависимости от типов столбцов. Однако создавать таблицы с таким большим числом столбцов обычно не требуется, а если такая потребность возникает, это скорее признак сомнительного дизайна.

Если таблица вам больше не нужна, вы можете удалить её, выполнив команду DROP TABLE. Например:

```
DROP TABLE my_first_table;
DROP TABLE products;
```

Попытка удаления несуществующей таблицы считается ошибкой. Тем не менее в SQL-скриптах часто применяют безусловное удаление таблиц перед созданием, игнорируя все сообщения об ошибках, так что они выполняют свою задачу независимо от того, существовали таблицы или нет. (Если вы хотите избежать таких ошибок, можно использовать вариант `DROP TABLE IF EXISTS`, но это не будет соответствовать стандарту SQL.)

Как изменить существующую таблицу, будет рассмотрено в этой главе позже, в Разделе 5.5.

Имея средства, которые мы обсудили, вы уже можете создавать полностью функциональные таблицы. В продолжении этой главы рассматриваются дополнительные возможности, призванные обеспечить целостность данных, безопасность и удобство. Если вам не терпится наполнить свои таблицы данными, вы можете вернуться к этой главе позже, а сейчас перейти к Главе 6.

5.2. Значения по умолчанию

Столбцу можно назначить значение по умолчанию. Когда добавляется новая строка и каким-то её столбцам не присваиваются значения, эти столбцы принимают значения по умолчанию. Также команда управления данными может явно указать, что столбцу должно быть присвоено значение по умолчанию, не зная его. (Подробнее команды управления данными описаны в Главе 6.)

Если значение по умолчанию не объявлено явно, им считается значение NULL. Обычно это имеет смысл, так как можно считать, что NULL представляет неизвестные данные.

В определении таблицы значения по умолчанию указываются после типа данных столбца. Например:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric DEFAULT 9.99
);
```

Значение по умолчанию может быть выражением, которое в этом случае вычисляется в момент присваивания значения по умолчанию (а *не* когда создаётся таблица). Например, столбцу `timestamp` в качестве значения по умолчания часто присваивается `CURRENT_TIMESTAMP`, чтобы в момент добавления строки в нём оказалось текущее время. Ещё один распространённый пример — генерация «последовательных номеров» для всех строк. В Postgres Pro это обычно делается примерно так:

```
CREATE TABLE products (
    product_no integer DEFAULT nextval('products_product_no_seq'),
    ...
);
```

здесь функция `nextval()` выбирает очередное значение из *последовательности* (см. Раздел 9.16). Это употребление настолько распространено, что для него есть специальная короткая запись:

```
CREATE TABLE products (
    product_no SERIAL,
    ...
);
```

`SERIAL` обсуждается позже в Подразделе 8.1.4.

5.3. Ограничения

Типы данных сами по себе ограничивают множество данных, которые можно сохранить в таблице. Однако для многих приложений такие ограничения слишком грубые. Например, столбец, содержащий цену продукта, должен, вероятно, принимать только положительные значения. Но такого стандартного типа данных нет. Возможно, вы также захотите ограничить данные столбца по отношению к другим столбцам или строкам. Например, в таблице с информацией о товаре должна быть только одна строка с определённым кодом товара.

Для решения подобных задач SQL позволяет вам определять ограничения для столбцов и таблиц. Ограничения дают вам возможность управлять данными в таблицах так, как вы захотите. Если пользователь попытается сохранить в столбце значение, нарушающее ограничения, возникнет ошибка. Ограничения будут действовать, даже если это значение по умолчанию.

5.3.1. Ограничения-проверки

Ограничение-проверка — наиболее общий тип ограничений. В его определении вы можете указать, что значение данного столбца должно удовлетворять логическому выражению (проверке истинности). Например, цену товара можно ограничить положительными значениями так:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0)
);
```

Как вы видите, ограничение определяется после типа данных, как и значение по умолчанию. Значения по умолчанию и ограничения могут указываться в любом порядке. Ограничение-проверка состоит из ключевого слова `CHECK`, за которым идёт выражение в скобках. Это выражение должно включать столбец, для которого задаётся ограничение, иначе оно не имеет большого смысла.

Вы можете также присвоить ограничению отдельное имя. Это улучшит сообщения об ошибках и позволит вам ссылаться на это ограничение, когда вам понадобится изменить его. Сделать это можно так:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CONSTRAINT positive_price CHECK (price > 0)
);
```

То есть, чтобы создать именованное ограничение, напишите ключевое слово `CONSTRAINT`, а за ним идентификатор и собственно определение ограничения. (Если вы не определите имя ограничения таким образом, система выберет для него имя за вас.)

Ограничение-проверка может также ссылаться на несколько столбцов. Например, если вы храните обычную цену и цену со скидкой, так вы можете гарантировать, что цена со скидкой будет всегда меньше обычной:

```
CREATE TABLE products (
    product_no integer,
    name text,
```

```
    price numeric CHECK (price > 0),
    discounted_price numeric CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

Первые два ограничения определяются похожим образом, но для третьего используется новый синтаксис. Оно не связано с определённым столбцом, а представлено отдельным элементом в списке. Определения столбцов и такие определения ограничений можно переставлять в произвольном порядке.

Про первые два ограничения можно сказать, что это ограничения столбцов, тогда как третье является ограничением таблицы, так как оно написано отдельно от определений столбцов. Ограничения столбцов также можно записать в виде ограничений таблицы, тогда как обратное не всегда возможно, так как подразумевается, что ограничение столбца ссылается только на связанный столбец. (Хотя Postgres Pro этого не требует, но для совместимости с другими СУБД лучше следовать это правилу.) Ранее приведённый пример можно переписать и так:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

Или даже так:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0 AND price > discounted_price)
);
```

Это дело вкуса.

Ограничениям таблицы можно присваивать имена так же, как и ограничениям столбцов:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CONSTRAINT valid_discount CHECK (price > discounted_price)
);
```

Следует заметить, что ограничение-проверка удовлетворяется, если выражение принимает значение true или NULL. Так как результатом многих выражений с операндами NULL будет значение NULL, такие ограничения не будут препятствовать записи NULL в связанные столбцы. Чтобы гарантировать, что столбец не содержит значения NULL, можно использовать ограничение NOT NULL, описанное в следующем разделе.

Примечание

Postgres Pro не поддерживает ограничения CHECK, которые обращаются к данным, не относящимся к новой или изменённой строке. Хотя ограничение CHECK, нарушающее

это правило, может работать в простых случаях, в общем случае нельзя гарантировать, что база данных не придёт в состояние, когда условие ограничения окажется ложным (вследствие последующих изменений других участвующих в его вычислении строк). В результате восстановление выгруженных данных может оказаться невозможным. Во время восстановления возможен сбой, даже если полное состояние базы данных согласуется с условием ограничения, по причине того, что строки загружаются не в том порядке, в котором это условие будет соблюдаться. Поэтому для определения ограничений, затрагивающих другие строки и другие таблицы, используйте ограничения `UNIQUE`, `EXCLUDE` или `FOREIGN KEY`, если это возможно.

Если вам не нужна постоянно поддерживаемая гарантия целостности, а достаточно разовой проверки добавляемой строки по отношению к другим строкам, вы можете реализовать эту проверку в собственном триггере. (Этот подход исключает вышеописанные проблемы при восстановлении, так как в выгрузке pg_dump триггеры воссоздаются после перезагрузки данных, и поэтому эта проверка не будет действовать в процессе восстановления.)

Примечание

В Postgres Pro предполагается, что условия ограничений `CHECK` являются постоянными, то есть при одинаковых данных в строке они всегда выдают одинаковый результат. Именно этим предположением оправдывается то, что ограничения `CHECK` проверяются только при добавлении или изменении строк, а не при каждом обращении к ним. (Приведённое выше предупреждение о недопустимости обращений к другим таблицам является частным следствием этого предположения.)

Однако это предположение может нарушаться, как часто бывает, когда в выражении `CHECK` используется пользовательская функция, поведение которой впоследствии меняется. Postgres Pro не запрещает этого, и если строки в таблице перестанут удовлетворять ограничению `CHECK`, это останется незамеченным. В итоге при попытке загрузить выгруженные позже данные могут возникнуть проблемы. Поэтому подобные изменения рекомендуется осуществлять следующим образом: удалить ограничение (используя `ALTER TABLE`), изменить определение функции, а затем пересоздать ограничение той же командой, которая при этом перепроверит все строки таблицы.

5.3.2. Ограничения NOT NULL

Ограничение NOT NULL просто указывает, что столбцу нельзя присваивать значение NULL. Пример синтаксиса:

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric
);
```

Ограничение NOT NULL всегда записывается как ограничение столбца и функционально эквивалентно ограничению `CHECK (`*`имя_столбца`*` IS NOT NULL)`, но в Postgres Pro явное ограничение NOT NULL работает более эффективно. Хотя у такой записи есть недостаток — назначить имя таким ограничениям нельзя.

Естественно, для столбца можно определить больше одного ограничения. Для этого их нужно просто указать одно за другим:

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric NOT NULL CHECK (price > 0)
```

```
);
```

Порядок здесь не имеет значения, он не обязательно соответствует порядку проверки ограничений.

Для ограничения `NOT NULL` есть и обратное: ограничение `NULL`. Оно не означает, что столбец должен иметь только значение NULL, что конечно было бы бессмысленно. Суть же его в простом указании, что столбец может иметь значение NULL (это поведение по умолчанию). Ограничение `NULL` отсутствует в стандарте SQL и использовать его в переносимых приложениях не следует. (Оно было добавлено в Postgres Pro только для совместимости с некоторыми другими СУБД.) Однако некоторые пользователи любят его использовать, так как оно позволяет легко переключать ограничения в скрипте. Например, вы можете начать с:

```
CREATE TABLE products (
    product_no integer NULL,
    name text NULL,
    price numeric NULL
);
```

и затем вставить ключевое слово `NOT`, где потребуется.

> **Подсказка**
>
> При проектировании баз данных чаще всего большинство столбцов должны быть помечены как NOT NULL.

5.3.3. Ограничения уникальности

Ограничения уникальности гарантируют, что данные в определённом столбце или группе столбцов уникальны среди всех строк таблицы. Ограничение записывается так:

```
CREATE TABLE products (
    product_no integer UNIQUE,
    name text,
    price numeric
);
```

в виде ограничения столбца и так:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE (product_no)
);
```

в виде ограничения таблицы.

Чтобы определить ограничение уникальности для группы столбцов, запишите его в виде ограничения таблицы, перечислив имена столбцов через запятую:

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c)
);
```

Такое ограничение указывает, что сочетание значений перечисленных столбцов должно быть уникально во всей таблице, тогда как значения каждого столбца по отдельности не должны быть (и обычно не будут) уникальными.

Вы можете назначить уникальному ограничению имя обычным образом:

```
CREATE TABLE products (
    product_no integer CONSTRAINT must_be_different UNIQUE,
    name text,
    price numeric
);
```

При добавлении ограничения уникальности будет автоматически создан уникальный индекс-B-дерево для столбца или группы столбцов, перечисленных в ограничении. Условие уникальности, распространяющееся только на некоторые строки, нельзя записать в виде ограничения уникальности, однако такое условие можно установить, создав уникальный частичный индекс.

Вообще говоря, ограничение уникальности нарушается, если в таблице оказывается несколько строк, у которых совпадают значения всех столбцов, включённых в ограничение. Однако два значения NULL при сравнении никогда не считаются равными. Это означает, что даже при наличии ограничения уникальности в таблице можно сохранить строки с дублирующимися значениями, если они содержат NULL в одном или нескольких столбцах ограничения. Это поведение соответствует стандарту SQL, но мы слышали о СУБД, которые ведут себя по-другому. Имейте в виду эту особенность, разрабатывая переносимые приложения.

5.3.4. Первичные ключи

Ограничение первичного ключа означает, что образующий его столбец или группа столбцов может быть уникальным идентификатором строк в таблице. Для этого требуется, чтобы значения были одновременно уникальными и отличными от NULL. Таким образом, таблицы со следующими двумя определениями будут принимать одинаковые данные:

```
CREATE TABLE products (
    product_no integer UNIQUE NOT NULL,
    name text,
    price numeric
);
```

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

Первичные ключи могут включать несколько столбцов; синтаксис похож на запись ограничений уникальности:

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (a, c)
);
```

При добавлении первичного ключа автоматически создаётся уникальный индекс-B-дерево для столбца или группы столбцов, перечисленных в первичном ключе, и данные столбцы помечаются как `NOT NULL`.

Таблица может иметь максимум один первичный ключ. (Ограничений уникальности и ограничений NOT NULL, которые функционально почти равнозначны первичным ключам, может быть сколько угодно, но назначить ограничением первичного ключа можно только одно.) Теория реляционных баз данных говорит, что первичный ключ должен быть в каждой таблице. В Postgres Pro такого жёсткого требования нет, но обычно лучше ему следовать.

Первичные ключи полезны и для документирования, и для клиентских приложений. Например, графическому приложению с возможностями редактирования содержимого таблицы, вероятно, потребуется знать первичный ключ таблицы, чтобы однозначно идентифицировать её строки.

Первичные ключи находят и другое применение в СУБД; в частности, первичный ключ в таблице определяет целевые столбцы по умолчанию для сторонних ключей, ссылающихся на эту таблицу.

5.3.5. Внешние ключи

Ограничение внешнего ключа указывает, что значения столбца (или группы столбцов) должны соответствовать значениям в некоторой строке другой таблицы. Это называется *ссылочной целостностью* двух связанных таблиц.

Пусть у вас уже есть таблица продуктов, которую мы неоднократно использовали ранее:

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

Давайте предположим, что у вас есть таблица с заказами этих продуктов. Мы хотим, чтобы в таблице заказов содержались только заказы действительно существующих продуктов. Поэтому мы определим в ней ограничение внешнего ключа, ссылающееся на таблицу продуктов:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products (product_no),
    quantity integer
);
```

С таким ограничением создать заказ со значением `product_no`, отсутствующим в таблице products (и не равным NULL), будет невозможно.

В такой схеме таблицу orders называют *подчинённой* таблицей, а products — *главной*. Соответственно, столбцы называют так же подчинённым и главным (или ссылающимся и целевым).

Предыдущую команду можно сократить так:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products,
    quantity integer
);
```

то есть, если опустить список столбцов, внешний ключ будет неявно связан с первичным ключом главной таблицы.

Ограничению внешнего ключа можно назначить имя стандартным способом.

Внешний ключ также может ссылаться на группу столбцов. В этом случае его нужно записать в виде обычного ограничения таблицы. Например:

```
CREATE TABLE t1 (
  a integer PRIMARY KEY,
  b integer,
  c integer,
  FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)
);
```

Естественно, число и типы столбцов в ограничении должны соответствовать числу и типам целевых столбцов.

Иногда имеет смысл задать в ограничении внешнего ключа в качестве «другой таблицы» ту же таблицу; такой внешний ключ называется *ссылающимся на себя*. Например, если вы хотите, чтобы строки таблицы представляли узлы древовидной структуры, вы можете написать

```
CREATE TABLE tree (
    node_id integer PRIMARY KEY,
    parent_id integer REFERENCES tree,
    name text,
    ...
);
```

Для узла верхнего уровня `parent_id` будет равен NULL, но записи с отличным от NULL `parent_id` будут ссылаться только на существующие строки таблицы.

Таблица может содержать несколько ограничений внешнего ключа. Это полезно для связи таблиц в отношении многие-ко-многим. Скажем, у вас есть таблицы продуктов и заказов, но вы хотите, чтобы один заказ мог содержать несколько продуктов (что невозможно в предыдущей схеме). Для этого вы можете использовать такую схему:

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products,
    order_id integer REFERENCES orders,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

Заметьте, что в последней таблице первичный ключ покрывает внешние ключи.

Мы знаем, что внешние ключи запрещают создание заказов, не относящихся ни к одному продукту. Но что делать, если после создания заказов с определённым продуктом мы захотим удалить его? SQL справится с этой ситуацией. Интуиция подсказывает следующие варианты поведения:

- Запретить удаление продукта
- Удалить также связанные заказы
- Что-то ещё?

Для иллюстрации давайте реализуем следующее поведение в вышеприведённом примере: при попытке удаления продукта, на который ссылаются заказы (через таблицу `order_items`), мы запрещаем эту операцию. Если же кто-то попытается удалить заказ, то удалится и его содержимое:

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);
```

```
CREATE TABLE order_items (
    product_no integer REFERENCES products ON DELETE RESTRICT,
    order_id integer REFERENCES orders ON DELETE CASCADE,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

Ограничивающие и каскадные удаления — два наиболее распространённых варианта. `RESTRICT` предотвращает удаление связанной строки. `NO ACTION` означает, что если зависимые строки продолжают существовать при проверке ограничения, возникает ошибка (это поведение по умолчанию). (Главным отличием этих двух вариантов является то, что `NO ACTION` позволяет отложить проверку в процессе транзакции, а `RESTRICT` — нет.) `CASCADE` указывает, что при удалении связанных строк зависимые от них будут так же автоматически удалены. Есть ещё два варианта: `SET NULL` и `SET DEFAULT`. При удалении связанных строк они назначают зависимым столбцам в подчинённой таблице значения NULL или значения по умолчанию, соответственно. Заметьте, что это не будет основанием для нарушения ограничений. Например, если в качестве действия задано `SET DEFAULT`, но значение по умолчанию не удовлетворяет ограничению внешнего ключа, операция закончится ошибкой.

Аналогично указанию `ON DELETE` существует `ON UPDATE`, которое срабатывает при изменении заданного столбца. При этом возможные действия те же, а `CASCADE` в данном случае означает, что изменённые значения связанных столбцов будут скопированы в зависимые строки.

Обычно зависимая строка не должна удовлетворять ограничению внешнего ключа, если один из связанных столбцов содержит NULL. Если в объявление внешнего ключа добавлено `MATCH FULL`, строка будет удовлетворять ограничению, только если все связанные столбцы равны NULL (то есть при разных значениях (NULL и не NULL) гарантируется невыполнение ограничения `MATCH FULL`). Если вы хотите, чтобы зависимые строки не могли избежать и этого ограничения, объявите связанные столбцы как `NOT NULL`.

Внешний ключ должен ссылаться на столбцы, образующие первичный ключ или ограничение уникальности. Таким образом, для связанных столбцов всегда будет существовать индекс (определённый соответствующим первичным ключом или ограничением), а значит проверки соответствия связанной строки будут выполняться эффективно. Так как команды `DELETE` для строк главной таблицы или `UPDATE` для зависимых столбцов потребуют просканировать подчинённую таблицу и найти строки, ссылающиеся на старые значения, полезно будет иметь индекс и для подчинённых столбцов. Но это нужно не всегда, и создать соответствующий индекс можно по-разному, поэтому объявление внешнего ключа не создаёт автоматически индекс по связанным столбцам.

Подробнее об изменении и удалении данных рассказывается в Главе 6. Вы также можете подробнее узнать о синтаксисе ограничений внешнего ключа в справке CREATE TABLE.

5.3.6. Ограничения-исключения

Ограничения-исключения гарантируют, что при сравнении любых двух строк по указанным столбцам или выражениям с помощью заданных операторов, минимум одно из этих сравнений возвратит false или NULL. Записывается это так:

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

Подробнее об этом см. `CREATE TABLE ... CONSTRAINT ... EXCLUDE`.

При добавлении ограничения-исключения будет автоматически создан индекс того типа, который указан в объявлении ограничения.

5.4. Системные столбцы

В каждой таблице есть несколько *системных столбцов*, неявно определённых системой. Как следствие, их имена нельзя использовать в качестве имён пользовательских столбцов. (Заметьте, что это не зависит от того, является ли имя ключевым словом или нет; заключение имени в кавычки не поможет избежать этого ограничения.) Эти столбцы не должны вас беспокоить, вам лишь достаточно знать об их существовании.

`oid`

Идентификатор объекта (object ID) для строки. Этот столбец присутствует, только если таблица была создана с указанием `WITH OIDS`, или если в момент её создания была установлена переменная конфигурации default_with_oids. Этот столбец имеет тип `oid` (с тем же именем, что и сам столбец); подробнее об этом типе см. Раздел 8.18.

`tableoid`

Идентификатор объекта для таблицы, содержащей строку. Этот столбец особенно полезен для запросов, имеющих дело с иерархией наследования (см. Раздел 5.9), так как без него сложно определить, из какой таблицы выбрана строка. Связав `tableoid` со столбцом `oid` в таблице `pg_class`, можно будет получить имя таблицы.

`xmin`

Идентификатор (код) транзакции, добавившей строку этой версии. (Версия строки — это её индивидуальное состояние; при каждом изменении создаётся новая версия одной и той же логической строки.)

`cmin`

Номер команды (начиная с нуля) внутри транзакции, добавившей строку.

`xmax`

Идентификатор транзакции, удалившей строку, или 0 для неудалённой версии строки. Значение этого столбца может быть ненулевым и для видимой версии строки. Это обычно означает, что удаляющая транзакция ещё не была зафиксирована, или удаление было отменено.

`cmax`

Номер команды в удаляющей транзакции или ноль.

`ctid`

Физическое расположение данной версии строки в таблице. Заметьте, что хотя по `ctid` можно очень быстро найти версию строки, значение `ctid` изменится при выполнении `VACUUM FULL`. Таким образом, `ctid` нельзя применять в качестве долгосрочного идентификатора строки. Для идентификации логических строк лучше использовать OID или даже дополнительный последовательный номер.

Коды OID представляют собой 32-битные значения и выбираются из единого для всей СУБД счётчика. В больших или долгоживущих базах данных этот счётчик может пойти по кругу. Таким образом, не рекомендуется рассчитывать на уникальность OID, если только вы не обеспечите её дополнительно. Если вам нужно идентифицировать строки таблицы, настоятельно рекомендуется использовать последовательности. Однако можно использовать и коды OID, при выполнении следующих условий:

- Когда для идентификации строк таблиц применяется OID, в каждой такой таблице должно создаваться ограничение уникальности для столбца OID. Когда такое ограничение уникальности (или уникальный индекс) существует, система позаботится о том, чтобы OID новой строки не совпал с уже существующими. (Конечно, это возможно, только если в таблице меньше 2^{32} (4 миллиардов) строк, а на практике таблицы должны быть гораздо меньше, иначе может пострадать производительность системы.)

- Никогда не следует рассчитывать, что OID будут уникальны среди всех таблиц; в качестве глобального идентификатора в рамках базы данных используйте комбинацию `tableoid` и OID строки.
- Конечно, все эти таблицы должны быть созданы с указанием `WITH OIDS`. В PostgreSQL 8.1 и новее по умолчанию подразумевается `WITHOUT OIDS`.

Идентификаторы транзакций также являются 32-битными. В долгоживущей базе данных они могут пойти по кругу. Это не критично при правильном обслуживании БД; подробнее об этом см. Главу 23. Однако полагаться на уникальность кодов транзакций в течение длительного времени (при более чем миллиарде транзакций) не следует.

Идентификаторы команд также 32-битные. Это создаёт жёсткий лимит на 2^{32} (4 миллиарда) команд SQL в одной транзакции. На практике это не проблема — заметьте, что это лимит числа команд SQL, а не количества обрабатываемых строк. Кроме того, идентификатор получают только те команды, которые фактически изменяют содержимое базы данных.

5.5. Изменение таблиц

Если вы создали таблицы, а затем поняли, что допустили ошибку, или изменились требования вашего приложения, вы можете удалить её и создать заново. Но это будет неудобно, если таблица уже заполнена данными, или если на неё ссылаются другие объекты базы данных (например, по внешнему ключу). Поэтому Postgres Pro предоставляет набор команд для модификации таблиц. Заметьте, что это по сути отличается от изменения данных, содержащихся в таблице: здесь мы обсуждаем модификацию определения, или структуры, таблицы.

Вы можете:

- Добавлять столбцы
- Удалять столбцы
- Добавлять ограничения
- Удалять ограничения
- Изменять значения по умолчанию
- Изменять типы столбцов
- Переименовывать столбцы
- Переименовывать таблицы

Все эти действия выполняются с помощью команды ALTER TABLE; подробнее о ней вы можете узнать в её справке.

5.5.1. Добавление столбца

Добавить столбец вы можете так:

```
ALTER TABLE products ADD COLUMN description text;
```

Новый столбец заполняется заданным для него значением по умолчанию (или значением NULL, если вы не добавите указание `DEFAULT`).

При этом вы можете сразу определить ограничения столбца, используя обычный синтаксис:

```
ALTER TABLE products ADD COLUMN description text CHECK (description <> '');
```

На самом деле здесь можно использовать все конструкции, допустимые в определении столбца в команде `CREATE TABLE`. Помните однако, что значение по умолчанию должно удовлетворять данным ограничениям, чтобы операция `ADD` выполнилась успешно. Вы также можете сначала заполнить столбец правильно, а затем добавить ограничения (см. ниже).

> **Подсказка**
>
> Добавление столбца со значением по умолчанию приводит к изменению всех строк таблицы (в них будет сохранено новое значение). Однако если значение по умолчанию не указано,

> Postgres Pro может обойтись без физического изменения. Поэтому, если вы планируете заполнить столбец в основном не значениями по умолчанию, лучше будет добавить столбец без значения по умолчанию, затем вставить требуемые значения с помощью `UPDATE`, а потом определить значение по умолчанию, как описано ниже.

5.5.2. Удаление столбца

Удалить столбец можно так:

```
ALTER TABLE products DROP COLUMN description;
```

Данные, которые были в этом столбце, исчезают. Вместе со столбцом удаляются и включающие его ограничения таблицы. Однако если на столбец ссылается ограничение внешнего ключа другой таблицы, Postgres Pro не удалит это ограничение неявно. Разрешить удаление всех зависящих от этого столбца объектов можно, добавив указание `CASCADE`:

```
ALTER TABLE products DROP COLUMN description CASCADE;
```

Общий механизм, стоящий за этим, описывается в Разделе 5.13.

5.5.3. Добавление ограничения

Для добавления ограничения используется синтаксис ограничения таблицы. Например:

```
ALTER TABLE products ADD CHECK (name <> '');
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);
ALTER TABLE products ADD FOREIGN KEY (product_group_id)
  REFERENCES product_groups;
```

Чтобы добавить ограничение NOT NULL, которое нельзя записать в виде ограничения таблицы, используйте такой синтаксис:

```
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;
```

Ограничение проходит проверку автоматически и будет добавлено, только если ему удовлетворяют данные таблицы.

5.5.4. Удаление ограничения

Для удаления ограничения вы должны знать его имя. Если вы не присваивали ему имя, это неявно сделала система, и вы должны выяснить его. Здесь может быть полезна команда psql `\d` *`имя_таблицы`* (или другие программы, показывающие подробную информацию о таблицах). Зная имя, вы можете использовать команду:

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

(Если вы имеете дело с именем ограничения вида `$2`, не забудьте заключить его в кавычки, чтобы это был допустимый идентификатор.)

Как и при удалении столбца, если вы хотите удалить ограничение с зависимыми объектами, добавьте указание `CASCADE`. Примером такой зависимости может быть ограничение внешнего ключа, связанное со столбцами ограничения первичного ключа.

Так можно удалить ограничения любых типов, кроме NOT NULL. Чтобы удалить ограничение NOT NULL, используйте команду:

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

(Вспомните, что у ограничений NOT NULL нет имён.)

5.5.5. Изменение значения по умолчанию

Назначить столбцу новое значение по умолчанию можно так:

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

Заметьте, что это никак не влияет на существующие строки таблицы, а просто задаёт значение по умолчанию для последующих команд INSERT.

Чтобы удалить значение по умолчанию, выполните:

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

При этом по сути значению по умолчанию просто присваивается NULL. Как следствие, ошибки не будет, если вы попытаетесь удалить значение по умолчанию, не определённое явно, так как неявно оно существует и равно NULL.

5.5.6. Изменение типа данных столбца

Чтобы преобразовать столбец в другой тип данных, используйте команду:

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

Она будет успешна, только если все существующие значения в столбце могут быть неявно приведены к новому типу. Если требуется более сложное преобразование, вы можете добавить указание USING, определяющее, как получить новые значения из старых.

Postgres Pro попытается также преобразовать к новому типу значение столбца по умолчанию (если оно определено) и все связанные с этим столбцом ограничения. Но преобразование может оказаться неправильным, и тогда вы получите неожиданные результаты. Поэтому обычно лучше удалить все ограничения столбца, перед тем как менять его тип, а затем воссоздать модифицированные должным образом ограничения.

5.5.7. Переименование столбца

Чтобы переименовать столбец, выполните:

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

5.5.8. Переименование таблицы

Таблицу можно переименовать так:

```
ALTER TABLE products RENAME TO items;
```

5.6. Права

Когда в базе данных создаётся объект, ему назначается владелец. Владельцем обычно становится роль, с которой был выполнен оператор создания. Для большинства типов объектов в исходном состоянии только владелец (или суперпользователь) может делать с объектом всё, что угодно. Чтобы разрешить использовать его другим ролям, нужно дать им *права*.

Существует несколько типов прав: SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER, CREATE, CONNECT, TEMPORARY, EXECUTE и USAGE. Набор прав, применимых к определённому объекту, зависит от типа объекта (таблица, функция и т. д.). Полную информацию о различных типах прав, поддерживаемых Postgres Pro, вы найдете на странице справки GRANT. Вы также увидите, как применяются эти права, в следующих разделах и главах.

Неотъемлемое право изменять или удалять объект имеет только владелец объекта.

Объекту можно назначить нового владельца с помощью команды ALTER для соответствующего типа объекта, например ALTER TABLE. Суперпользователь может делать это без ограничений, а обычный пользователь, только если он является одновременно текущим владельцем объекта (или членом роли владельца) и членом новой роли.

Для назначения прав применяется команда GRANT. Например, если в базе данных есть роль joe и таблица accounts, право на изменение таблицы можно дать этой роли так:

```
GRANT UPDATE ON accounts TO joe;
```

Если вместо конкретного права написать ALL, роль получит все права, применимые для объекта этого типа.

Для назначения права всем ролям в системе можно использовать специальное имя «роли»: PUBLIC. Также для упрощения управления ролями, когда в базе данных есть множество пользователей, можно настроить «групповые» роли; подробнее об этом см. Главу 20.

Чтобы лишить пользователей прав, используйте команду REVOKE:

```
REVOKE ALL ON accounts FROM PUBLIC;
```

Особые права владельца объекта (то есть права на выполнение DROP, GRANT, REVOKE и т. д.) всегда неявно закреплены за владельцем и их нельзя назначить или отобрать. Но владелец объекта может лишить себя обычных прав, например, разрешить всем, включая себя, только чтение таблицы.

Обычно распоряжаться правами может только владелец объекта (или суперпользователь). Однако возможно дать право доступа к объекту «с правом передачи», что позволит получившему такое право назначать его другим. Если такое право передачи впоследствии будет отозвано, то все, кто получил данное право доступа (непосредственно или по цепочке передачи), потеряют его. Подробнее об этом см. справку GRANT и REVOKE.

5.7. Политики защиты строк

В дополнение к стандартной системе прав SQL, управляемой командой GRANT, на уровне таблиц можно определить *политики защиты строк*, ограничивающие для пользователей наборы строк, которые могут быть возвращены обычными запросами или добавлены, изменены и удалены командами, изменяющими данные. Это называется также *защитой на уровне строк* (RLS, Row-Level Security). По умолчанию таблицы не имеют политик, так что если система прав SQL разрешает пользователю доступ к таблице, все строки в ней одинаково доступны для чтения или изменения.

Когда для таблицы включается защита строк (с помощью команды ALTER TABLE ... ENABLE ROW LEVEL SECURITY), все обычные запросы к таблице на выборку или модификацию строк должны разрешаться политикой защиты строк. (Однако на владельца таблицы такие политики обычно не действуют.) Если политика для таблицы не определена, применяется политика запрета по умолчанию, так что никакие строки в этой таблице нельзя увидеть или модифицировать. На операции с таблицей в целом, такие как TRUNCATE и REFERENCES, защита строк не распространяется.

Политики защиты строк могут применяться к определённым командам и/или ролям. Политику можно определить как применяемую к командам ALL (всем), либо SELECT, INSERT, UPDATE и DELETE. Кроме того, политику можно связать с несколькими ролями, при этом действуют обычные правила членства и наследования.

Чтобы определить, какие строки будут видимыми или могут изменяться в таблице, для политики задаётся выражение, возвращающее логический результат. Это выражение будет вычисляться для каждой строки перед другими условиями или функциями, поступающими из запроса пользователя. (Единственным исключением из этого правила являются герметичные функции, которые гарантированно не допускают утечки информации; оптимизатор может решить выполнить эти функции до проверок защиты строк.) Строки, для которых это выражение возвращает не true, обрабатываться не будут. Чтобы независимо управлять набором строк, которые можно видеть, и набором строк, которые можно модифицировать, в политике можно задать отдельные выражения. Выражения политик обрабатываются в составе запроса с правами исполняющего его пользователя, но для обращения к данным, недоступным этому пользователю, в этих выражениях могут применяться функции, определяющие контекст безопасности.

Суперпользователи и роли с атрибутом BYPASSRLS всегда обращаются к таблице, минуя систему защиты строк. На владельца таблицы защита строк тоже не действует, хотя он может включить её для себя принудительно, выполнив команду ALTER TABLE ... FORCE ROW LEVEL SECURITY.

Неотъемлемое право включать или отключать защиту строк, а также определять политики для таблицы, имеет только её владелец.

Для создания политик предназначена команда CREATE POLICY, для изменения — ALTER POLICY, а для удаления — DROP POLICY. Чтобы включить или отключить защиту строк для определённой таблицы, воспользуйтесь командой ALTER TABLE.

Каждой политике назначается имя, при этом для одной таблицы можно определить несколько политик. Так как политики привязаны к таблицам, каждая политика для таблицы должна иметь уникальное имя. В разных таблицах политики могут иметь одинаковые имена.

Когда к определённому запросу применяются несколько политик, они объединяются либо логическим сложением (если политики разрешительные (по умолчанию)), либо умножением (если политики ограничительные). Это подобно тому, как некоторая роль получает права всех ролей, в которые она включена. Разрешительные и ограничительные политики рассматриваются ниже.

В качестве простого примера, создать политику для отношения `account`, позволяющую только членам роли `managers` обращаться к строкам отношения и при этом только к своим, можно так:

```
CREATE TABLE accounts (manager text, company text, contact_email text);

ALTER TABLE accounts ENABLE ROW LEVEL SECURITY;

CREATE POLICY account_managers ON accounts TO managers
    USING (manager = current_user);
```

Эта политика неявно подразумевает и предложение `WITH CHECK`, идентичное предложению `USING`, поэтому указанное ограничение применяется и к строкам, выбираемым командой (так что один менеджер не может выполнить `SELECT`, `UPDATE` или `DELETE` для существующих строк, принадлежащих другому), и к строкам, изменяемым командой (так что командами `INSERT` и `UPDATE` нельзя создать строки, принадлежащие другому менеджеру).

Если роль не задана, либо задано специальное имя пользователя `PUBLIC`, политика применяется ко всем пользователям в данной системе. Чтобы все пользователи могли обратиться только к собственной строке в таблице `users`, можно применить простую политику:

```
CREATE POLICY user_policy ON users
    USING (user_name = current_user);
```

Это работает подобно предыдущему примеру.

Чтобы определить для строк, добавляемых в таблицу, отдельную политику, отличную от политики, ограничивающей видимые строки, можно скомбинировать несколько политик. Следующая пара политик позволит всем пользователям видеть все строки в таблице `users`, но изменять только свою собственную:

```
CREATE POLICY user_sel_policy ON users
    FOR SELECT
    USING (true);
CREATE POLICY user_mod_policy ON users
    USING (user_name = current_user);
```

Для команды `SELECT` эти две политики объединяются операцией `OR`, так что в итоге это позволяет выбирать все строки. Для команд других типов применяется только вторая политика, и эффект тот же, что и раньше.

Защиту строк можно отключить так же командой `ALTER TABLE`. При отключении защиты, политики, определённые для таблицы, не удаляются, а просто игнорируются. В результате в таблице будут видны и могут модифицироваться все строки, с учётом ограничений стандартной системы прав SQL.

Ниже показан развёрнутый пример того, как этот механизм защиты можно применять в производственной среде. Таблица `passwd` имитирует файл паролей в Unix:

```
-- Простой пример на базе файла passwd
CREATE TABLE passwd (
  user_name              text UNIQUE NOT NULL,
  pwhash                 text,
  uid                    int  PRIMARY KEY,
  gid                    int  NOT NULL,
  real_name              text NOT NULL,
  home_phone             text,
  extra_info             text,
  home_dir               text NOT NULL,
  shell                  text NOT NULL
);

CREATE ROLE admin;  -- Администратор
CREATE ROLE bob;    -- Обычный пользователь
CREATE ROLE alice;  -- Обычный пользователь

-- Наполнение таблицы
INSERT INTO passwd VALUES
  ('admin','xxx',0,0,'Admin','111-222-3333',null,'/root','/bin/dash');
INSERT INTO passwd VALUES
  ('bob','xxx',1,1,'Bob','123-456-7890',null,'/home/bob','/bin/zsh');
INSERT INTO passwd VALUES
  ('alice','xxx',2,1,'Alice','098-765-4321',null,'/home/alice','/bin/zsh');

-- Необходимо включить для этой таблицы защиту на уровне строк
ALTER TABLE passwd ENABLE ROW LEVEL SECURITY;

-- Создание политик
-- Администратор может видеть и добавлять любые строки
CREATE POLICY admin_all ON passwd TO admin USING (true) WITH CHECK (true);
-- Обычные пользователи могут видеть все строки
CREATE POLICY all_view ON passwd FOR SELECT USING (true);
-- Обычные пользователи могут изменять собственные данные, но
-- не могут задать произвольную оболочку входа
CREATE POLICY user_mod ON passwd FOR UPDATE
  USING (current_user = user_name)
  WITH CHECK (
    current_user = user_name AND
    shell IN ('/bin/bash','/bin/sh','/bin/dash','/bin/zsh','/bin/tcsh')
  );

-- Администраторы получают все обычные права
GRANT SELECT, INSERT, UPDATE, DELETE ON passwd TO admin;
-- Пользователям разрешается чтение только общедоступных столбцов
GRANT SELECT
  (user_name, uid, gid, real_name, home_phone, extra_info, home_dir, shell)
  ON passwd TO public;
-- Пользователям разрешается изменение определённых столбцов
GRANT UPDATE
  (pwhash, real_name, home_phone, extra_info, shell)
  ON passwd TO public;
```

Как и любые средства защиты, важно проверить политики, и убедиться в том, что они работают ожидаемым образом. Применительно к предыдущему примеру, эти команды показывают, что система разрешений работает корректно.

```
-- Администратор может видеть все строки и поля
postgres=> set role admin;
```

```
SET
postgres=> table passwd;
 user_name | pwhash | uid | gid | real_name |  home_phone  | extra_info | home_dir    |
   shell
-----------+--------+-----+-----+-----------+--------------+------------+-------------
+-----------
 admin     | xxx    |   0 |   0 | Admin     | 111-222-3333 |            | /root
 | /bin/dash
 bob       | xxx    |   1 |   1 | Bob       | 123-456-7890 |            | /home/bob
 | /bin/zsh
 alice     | xxx    |   2 |   1 | Alice     | 098-765-4321 |            | /home/alice
 | /bin/zsh
(3 rows)

-- Проверим, что может делать Алиса
postgres=> set role alice;
SET
postgres=> table passwd;
ERROR:  permission denied for relation passwd
postgres=> select user_name,real_name,home_phone,extra_info,home_dir,shell from passwd;
 user_name | real_name |  home_phone  | extra_info | home_dir    |   shell
-----------+-----------+--------------+------------+-------------+-----------
 admin     | Admin     | 111-222-3333 |            | /root       | /bin/dash
 bob       | Bob       | 123-456-7890 |            | /home/bob   | /bin/zsh
 alice     | Alice     | 098-765-4321 |            | /home/alice | /bin/zsh
(3 rows)

postgres=> update passwd set user_name = 'joe';
ERROR:  permission denied for relation passwd
-- Алиса может изменить своё имя (поле real_name), но не имя кого-либо другого
postgres=> update passwd set real_name = 'Alice Doe';
UPDATE 1
postgres=> update passwd set real_name = 'John Doe' where user_name = 'admin';
UPDATE 0
postgres=> update passwd set shell = '/bin/xx';
ERROR:  new row violates WITH CHECK OPTION for "passwd"
postgres=> delete from passwd;
ERROR:  permission denied for relation passwd
postgres=> insert into passwd (user_name) values ('xxx');
ERROR:  permission denied for relation passwd
-- Алиса может изменить собственный пароль; попытки поменять другие пароли RLS просто
 игнорирует
postgres=> update passwd set pwhash = 'abc';
UPDATE 1
```

Все политики, создаваемые до этого, были разрешительными, что значит, что при применении нескольких политик они объединялись логическим оператором «ИЛИ». Хотя можно создать такие разрешительные политики, которые будут только разрешать доступ к строкам при определённых условиях, может быть проще скомбинировать разрешительные политики с ограничительными (которым должны удовлетворять записи и которые объединяются логическим оператором «И»). В развитие предыдущего примера мы добавим ограничительную политику, разрешающую администратору, подключённому через локальный сокет Unix, обращаться к записям таблицы `passwd`:

```
CREATE POLICY admin_local_only ON passwd AS RESTRICTIVE TO admin
    USING (pg_catalog.inet_client_addr() IS NULL);
```

Затем мы можем убедиться, что администратор, подключённый по сети, не увидит никаких записей, благодаря этой ограничительной политике:

```
=> SELECT current_user;
 current_user
--------------
 admin
(1 row)

=> select inet_client_addr();
 inet_client_addr
------------------
 127.0.0.1
(1 row)

=> SELECT current_user;
 current_user
--------------
 admin
(1 row)

=> TABLE passwd;
 user_name | pwhash | uid | gid | real_name | home_phone | extra_info | home_dir |
 shell
-----------+--------+-----+-----+-----------+------------+------------+----------
+-------
(0 rows)

=> UPDATE passwd set pwhash = NULL;
UPDATE 0
```

На проверки ссылочной целостности, например, на ограничения уникальности и внешние ключи, защита строк никогда не распространяется, чтобы не нарушалась целостность данных. Поэтому организацию и политики защиты на уровне строк необходимо тщательно прорабатывать, чтобы не возникли «скрытые каналы» утечки информации через эти проверки.

В некоторых случаях важно, чтобы защита на уровне строк, наоборот, не действовала. Например, резервное копирование может оказаться провальным, если механизм защиты на уровне строк молча не даст скопировать какие-либо строки. В таком случае вы можете установить для параметра конфигурации row_security значение `off`. Это само по себе не отключит защиту строк; при этом просто будет выдана ошибка, если результаты запроса отфильтруются политикой, с тем чтобы можно было изучить причину ошибки и устранить её.

В приведённых выше примерах выражения политики учитывали только текущие значения в запрашиваемой или изменяемой строке. Это самый простой и наиболее эффективный по скорости вариант; по возможности реализацию защиты строк следует проектировать именно так. Если же для принятия решения о доступе необходимо обращаться к другим строкам или другим таблицам, это можно осуществить, применяя в выражениях политик вложенные `SELECT` или функции, содержащие `SELECT`. Однако учтите, что при такой реализации возможны условия гонки, что чревато утечкой информации, если не принять меры предосторожности. Например, рассмотрим следующую конструкцию таблиц:

```
-- определение групп привилегий
CREATE TABLE groups (group_id int PRIMARY KEY,
                     group_name text NOT NULL);

INSERT INTO groups VALUES
  (1, 'low'),
  (2, 'medium'),
  (5, 'high');

GRANT ALL ON groups TO alice;  -- alice является администратором
```

```
GRANT SELECT ON groups TO public;

-- определение уровней привилегий для пользователей
CREATE TABLE users (user_name text PRIMARY KEY,
                    group_id int NOT NULL REFERENCES groups);

INSERT INTO users VALUES
  ('alice', 5),
  ('bob', 2),
  ('mallory', 2);

GRANT ALL ON users TO alice;
GRANT SELECT ON users TO public;

-- таблица, содержащая защищаемую информацию
CREATE TABLE information (info text,
                          group_id int NOT NULL REFERENCES groups);

INSERT INTO information VALUES
  ('barely secret', 1),
  ('slightly secret', 2),
  ('very secret', 5);

ALTER TABLE information ENABLE ROW LEVEL SECURITY;

-- строка должна быть доступна для чтения/изменения пользователям с group_id,
-- большим или равным group_id данной строки
CREATE POLICY fp_s ON information FOR SELECT
  USING (group_id <= (SELECT group_id FROM users WHERE user_name = current_user));
CREATE POLICY fp_u ON information FOR UPDATE
  USING (group_id <= (SELECT group_id FROM users WHERE user_name = current_user));

-- мы защищаем таблицу с информацией, полагаясь только на RLS
GRANT ALL ON information TO public;
```

Теперь предположим, что Алиса (роль `alice`) желает записать «слегка секретную» информацию, но при этом не хочет давать `mallory` доступ к ней. Она делает следующее:

```
BEGIN;
UPDATE users SET group_id = 1 WHERE user_name = 'mallory';
UPDATE information SET info = 'secret from mallory' WHERE group_id = 2;
COMMIT;
```

На первый взгляд всё нормально; `mallory` ни при каких условиях не должна увидеть строку «secret from mallory». Однако здесь возможно условие гонки. Если Мэллори (роль `mallory`) параллельно выполняет, скажем:

```
SELECT * FROM information WHERE group_id = 2 FOR UPDATE;
```

и её транзакция в режиме `READ COMMITTED`, она сможет увидеть «secret from mallory». Это произойдёт, если её транзакция дойдёт до строки `information` сразу после того, как эту строку изменит Алиса (роль `alice`). Она заблокируется, ожидая фиксирования транзакции Алисы, а затем прочитает изменённое содержимое строки благодаря предложению `FOR UPDATE`. Однако при этом изменённое содержимое `users` *не* будет прочитано неявным запросом `SELECT`, так как этот вложенный `SELECT` выполняется без указания `FOR UPDATE`; вместо этого строка `users` читается из снимка, полученного в начале запроса. Таким образом, выражение политики проверяет старое значение уровня привилегий пользователя `mallory` и позволяет ей видеть изменённую строку.

Обойти эту проблему можно несколькими способами. Первое простое решение заключается в использовании `SELECT ... FOR SHARE` во вложенных запросах `SELECT` в политиках защиты

строк. Однако для этого потребуется давать затронутым пользователям право UPDATE в целевой таблице (здесь users), что может быть нежелательно. (Хотя можно применить ещё одну политику защиты строк, чтобы они не могли практически воспользоваться этим правилом; либо поместить вложенный SELECT в функцию, определяющую контекст безопасности.) Кроме этого, активное использование блокировок строк в целевой таблице может повлечь проблемы с производительностью, особенно при частых изменениях. Другое решение, практичное, если целевая таблица изменяется нечасто, заключается в блокировке целевой таблицы в режиме ACCESS EXCLUSIVE при изменении, чтобы никакие параллельные транзакции не видели старые значения строк. Либо можно просто дождаться завершения всех параллельных транзакций после изменения в целевой таблице, прежде чем вносить изменения, рассчитанные на новые условия безопасности.

За дополнительными подробностями обратитесь к CREATE POLICY и ALTER TABLE.

5.8. Схемы

Кластер баз данных Postgres Pro содержит один или несколько именованных экземпляров баз. На уровне кластера создаются роли и некоторые другие объекты. При этом в рамках одного подключения к серверу можно обращаться к данным только одной базы — той, что была выбрана при установлении соединения.

> **Примечание**
>
> Пользователи кластера не обязательно будут иметь доступ ко всем базам данных этого кластера. Тот факт, что роли создаются на уровне кластера, означает только то, что в кластере не может быть двух ролей joe в разных базах данных, хотя система позволяет ограничить доступ joe только некоторыми базами данных.

База данных содержит одну или несколько именованных *схем*, которые в свою очередь содержат таблицы. Схемы также содержат именованные объекты других видов, включая типы данных, функции и операторы. Одно и то же имя объекта можно свободно использовать в разных схемах, например и schema1, и myschema могут содержать таблицы с именем mytable. В отличие от баз данных, схемы не ограничивают доступ к данным: пользователи могут обращаться к объектам в любой схеме текущей базы данных, если им назначены соответствующие права.

Есть несколько возможных объяснений, для чего стоит применять схемы:

- Чтобы одну базу данных могли использовать несколько пользователей, независимо друг от друга.
- Чтобы объединить объекты базы данных в логические группы для облегчения управления ими.
- Чтобы в одной базе сосуществовали разные приложения, и при этом не возникало конфликтов имён.

Схемы в некотором смысле подобны каталогам в операционной системе, но они не могут быть вложенными.

5.8.1. Создание схемы

Для создания схемы используется команда CREATE SCHEMA. При этом вы определяете имя схемы по своему выбору, например так:

```
CREATE SCHEMA myschema;
```

Чтобы создать объекты в схеме или обратиться к ним, указывайте *полное имя*, состоящее из имён схемы и объекта, разделённых точкой:

```
схема.таблица
```

Этот синтаксис работает везде, где ожидается имя таблицы, включая команды модификации таблицы и команды обработки данных, обсуждаемые в следующих главах. (Для краткости мы будем говорить только о таблицах, но всё это распространяется и на другие типы именованных объектов, например, типы и функции.)

Есть ещё более общий синтаксис

база_данных.*схема*.*таблица*

но в настоящее время он поддерживается только для формального соответствия стандарту SQL. Если вы указываете базу данных, это может быть только база данных, к которой вы подключены.

Таким образом, создать таблицу в новой схеме можно так:

```
CREATE TABLE myschema.mytable (
 ...
);
```

Чтобы удалить пустую схему (не содержащую объектов), выполните:

```
DROP SCHEMA myschema;
```

Удалить схему со всеми содержащимися в ней объектами можно так:

```
DROP SCHEMA myschema CASCADE;
```

Стоящий за этим общий механизм описан в Разделе 5.13.

Часто бывает нужно создать схему, владельцем которой будет другой пользователь (это один из способов ограничения пользователей пространствами имён). Сделать это можно так:

```
CREATE SCHEMA имя_схемы AUTHORIZATION имя_пользователя;
```

Вы даже можете опустить имя схемы, в этом случае именем схемы станет имя пользователя. Как это можно применять, описано в Подразделе 5.8.6.

Схемы с именами, начинающимися с `pg_`, являются системными; пользователям не разрешено использовать такие имена.

5.8.2. Схема public

До этого мы создавали таблицы, не указывая никакие имена схем. По умолчанию такие таблицы (и другие объекты) автоматически помещаются в схему «public». Она содержится во всех создаваемых базах данных. Таким образом, команда:

```
CREATE TABLE products ( ... );
```

эквивалентна:

```
CREATE TABLE public.products ( ... );
```

5.8.3. Путь поиска схемы

Везде писать полные имена утомительно, и часто всё равно лучше не привязывать приложения к конкретной схеме. Поэтому к таблицам обычно обращаются по *неполному имени*, состоящему просто из имени таблицы. Система определяет, какая именно таблица подразумевается, используя *путь поиска*, который представляет собой список просматриваемых схем. Подразумеваемой таблицей считается первая подходящая таблица, найденная в схемах пути. Если подходящая таблица не найдена, возникает ошибка, даже если таблица с таким именем есть в других схемах базы данных.

Возможность создавать одноимённые объекты в разных схемах усложняет написание запросов, которые должны всегда обращаться к конкретным объектам. Это также потенциально позволяет пользователям влиять на поведение запросов других пользователей, злонамеренно или случайно. Ввиду преобладания неполных имён в запросах и их использования внутри Postgres Pro, добавить схему в `search_path` — по сути значит доверять всем пользователям, имеющим право `CREATE` в этой схеме. Когда вы выполняете обычный запрос, злонамеренный пользователь может создать

объекты в схеме, включённой в ваш путь поиска, и таким образом перехватывать управление и выполнять произвольные функции SQL как если бы их выполняли вы.

Первая схема в пути поиска называется текущей. Эта схема будет использоваться не только при поиске, но и при создании объектов — она будет включать таблицы, созданные командой `CREATE TABLE` без указания схемы.

Чтобы узнать текущий тип поиска, выполните следующую команду:

```
SHOW search_path;
```

В конфигурации по умолчанию она возвращает:

```
 search_path
--------------
 "$user", public
```

Первый элемент ссылается на схему с именем текущего пользователя. Если такой схемы не существует, ссылка на неё игнорируется. Второй элемент ссылается на схему public, которую мы уже видели.

Первая существующая схема в пути поиска также считается схемой по умолчанию для новых объектов. Именно поэтому по умолчанию объекты создаются в схеме public. При указании неполной ссылки на объект в любом контексте (при модификации таблиц, изменении данных или в запросах) система просматривает путь поиска, пока не найдёт соответствующий объект. Таким образом, в конфигурации по умолчанию неполные имена могут относиться только к объектам в схеме public.

Чтобы добавить в путь нашу новую схему, мы выполняем:

```
SET search_path TO myschema,public;
```

(Мы опускаем компонент `$user`, так как здесь в нём нет необходимости.) Теперь мы можем обращаться к таблице без указания схемы:

```
DROP TABLE mytable;
```

И так как `myschema` — первый элемент в пути, новые объекты будут по умолчанию создаваться в этой схеме.

Мы можем также написать:

```
SET search_path TO myschema;
```

Тогда мы больше не сможем обращаться к схеме public, не написав полное имя объекта. Единственное, что отличает схему public от других, это то, что она существует по умолчанию, хотя её так же можно удалить.

В Разделе 9.25 вы узнаете, как ещё можно манипулировать путём поиска схем.

Как и для имён таблиц, путь поиска аналогично работает для имён типов данных, имён функций и имён операторов. Имена типов данных и функций можно записать в полном виде так же, как и имена таблиц. Если же вам нужно использовать в выражении полное имя оператора, для этого есть специальный способ — вы должны написать:

```
OPERATOR(схема.оператор)
```

Такая запись необходима для избежания синтаксической неоднозначности. Пример такого выражения:

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

На практике пользователи часто полагаются на путь поиска, чтобы не приходилось писать такие замысловатые конструкции.

5.8.4. Схемы и права

По умолчанию пользователь не может обращаться к объектам в чужих схемах. Чтобы изменить это, владелец схемы должен дать пользователю право USAGE для данной схемы. Чтобы пользователи могли использовать объекты схемы, может понадобиться назначить дополнительные права на уровне объектов.

Пользователю также можно разрешить создавать объекты в схеме, не принадлежащей ему. Для этого ему нужно дать право CREATE в требуемой схеме. Заметьте, что по умолчанию все имеют права CREATE и USAGE в схеме public. Благодаря этому все пользователи могут подключаться к заданной базе данных и создавать объекты в её схеме public. Некоторые шаблоны использования требуют запретить это:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

(Первое слово «public» обозначает схему, а второе означает «каждый пользователь». В первом случае это идентификатор, а во втором — ключевое слово, поэтому они написаны в разном регистре; вспомните указания из Подраздела 4.1.1.)

5.8.5. Схема системного каталога

В дополнение к схеме public и схемам, создаваемым пользователями, любая база данных содержит схему pg_catalog, в которой находятся системные таблицы и все встроенные типы данных, функции и операторы. pg_catalog фактически всегда является частью пути поиска. Если даже эта схема не добавлена в путь явно, она неявно просматривается *до* всех схем, указанных в пути. Так обеспечивается доступность встроенных имён при любых условиях. Однако вы можете явным образом поместить pg_catalog в конец пути поиска, если вам нужно, чтобы пользовательские имена переопределяли встроенные.

Так как имена системных таблиц начинаются с pg_, такие имена лучше не использовать во избежание конфликта имён, возможного при появлении в будущем системной таблицы с тем же именем, что и ваша. (С путём поиска по умолчанию неполная ссылка будет воспринята как обращение к системной таблице.) Системные таблицы будут и дальше содержать в имени приставку pg_, так что они не будут конфликтовать с неполными именами пользовательских таблиц, если пользователи со своей стороны не будут использовать приставку pg_.

5.8.6. Шаблоны использования

Схемам можно найти множество применений. Для защиты от влияния недоверенных пользователей на поведение запросов других пользователей предлагается *шаблон безопасного использования схем*, но если этот шаблон не применяется в базе данных, пользователи, желающие безопасно выполнять в ней запросы, должны будут принимать защитные меры в начале каждого сеанса. В частности, они должны начинать каждый сеанс с присваивания пустого значения переменной search_path или каким-либо другим образом удалять из search_path схемы, доступные для записи обычным пользователям. С конфигурацией по умолчанию легко реализуются следующие шаблоны использования:

- Ограничить обычных пользователей личными схемами. Для реализации этого подхода выполните REVOKE CREATE ON SCHEMA public FROM PUBLIC и создайте для каждого пользователя схему с его именем. Как вы знаете, путь поиска по умолчанию начинается с имени $user, вместо которого подставляется имя пользователя. Таким образом, если у всех пользователей будет отдельная схема, они по умолчанию будут обращаться к собственным схемам. Применяя этот шаблон в базе, к которой уже могли подключаться недоверенные пользователи, проверьте, нет ли в схеме public объектов с такими же именами, как у объектов в схеме pg_catalog. Этот шаблон является шаблоном безопасного использования схем, только если никакой недоверенный пользователь не является владельцем базы данных и не имеет права CREATEROLE. В противном случае безопасное использование схем невозможно.

- Удалить схему public из пути поиска по умолчанию, изменив postgresql.conf или выполнив команду ALTER ROLE ALL SET search_path = "$user". При этом все по-прежнему смогут создавать объекты в общей схеме, но выбираться эти объекты будут только по полному имени, со схемой. Тогда как обращаться к таблицам по полному имени вполне допустимо, обращения

к функциям в общей схеме всё же будут небезопасными или ненадёжными. Поэтому если вы создаёте функции или расширения в схеме public, применяйте первый шаблон. Если же нет, этот шаблон, как и первый, безопасен при условии, что никакой недоверенный пользователь не является владельцем базы данных и не имеет права CREATEROLE.

- Сохранить поведение по умолчанию. Все пользователи неявно обращаются к схеме public. Тем самым имитируется ситуация с полным отсутствием схем, что позволяет осуществить плавный переход из среды без схем. Однако данный шаблон ни в коем случае нельзя считать безопасным. Он подходит, только если в базе данных имеется всего один либо несколько доверяющих друг другу пользователей.

При любом подходе, устанавливая совместно используемые приложения (таблицы, которые нужны всем, дополнительные функции сторонних разработчиков и т. д.), помещайте их в отдельные схемы. Не забудьте дать другим пользователям права для доступа к этим схемам. Тогда пользователи смогут обращаться к этим дополнительным объектам по полному имени или при желании добавят эти схемы в свои пути поиска.

5.8.7. Переносимость

Стандарт SQL не поддерживает обращение в одной схеме к разным объектам, принадлежащим разным пользователям. Более того, в ряде реализаций СУБД нельзя создавать схемы с именем, отличным от имени владельца. На практике, в СУБД, реализующих только базовую поддержку схем согласно стандарту, концепции пользователя и схемы очень близки. Таким образом, многие пользователи полагают, что полное имя на самом деле образуется как *имя_пользователя*.*имя_таблицы*. И именно так будет вести себя Postgres Pro, если вы создадите схемы для каждого пользователя.

В стандарте SQL нет и понятия схемы public. Для максимального соответствия стандарту использовать схему public не следует.

Конечно, есть СУБД, в которых вообще не реализованы схемы или пространства имён поддерживают (возможно, с ограничениями) обращения к другим базам данных. Если вам потребуется работать с этими системами, максимальной переносимости вы достигнете, вообще не используя схемы.

5.9. Наследование

Postgres Pro реализует наследование таблиц, что может быть полезно для проектировщиков баз данных. (Стандарт SQL:1999 и более поздние версии определяют возможность наследования типов, но это во многом отличается от того, что описано здесь.)

Давайте начнём со следующего примера: предположим, что мы создаём модель данных для городов. В каждом штате есть множество городов, но лишь одна столица. Мы хотим иметь возможность быстро получать город-столицу для любого штата. Это можно сделать, создав две таблицы: одну для столиц штатов, а другую для городов, не являющихся столицами. Однако что делать, если нам нужно получить информацию о любом городе, будь то столица штата или нет? В решении этой проблемы может помочь наследование. Мы определим таблицу capitals как наследника cities:

```
CREATE TABLE cities (
    name            text,
    population      float,
    elevation       int     -- в футах
);

CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

В этом случае таблица capitals *наследует* все столбцы своей родительской таблицы, cities. Столицы штатов также имеют дополнительный столбец state, в котором будет указан штат.

В Postgres Pro таблица может наследоваться от нуля или нескольких других таблиц, а запросы могут выбирать все строки родительской таблицы или все строки родительской и всех дочерних таблиц. По умолчанию принят последний вариант. Например, следующий запрос найдёт названия всех городов, включая столицы штатов, расположенных выше 500 футов:

```
SELECT name, elevation
    FROM cities
    WHERE elevation > 500;
```

Для данных из введения (см. Раздел 2.1) он выдаст:

```
   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
 Madison   |       845
```

А следующий запрос находит все города, которые не являются столицами штатов, но также находятся на высоте выше 500 футов:

```
SELECT name, elevation
    FROM ONLY cities
    WHERE elevation > 500;

   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
```

Здесь ключевое слово `ONLY` указывает, что запрос должен применяться только к таблице `cities`, но не к таблицам, расположенным ниже `cities` в иерархии наследования. Многие операторы, которые мы уже обсудили, — `SELECT`, `UPDATE` и `DELETE` — поддерживают ключевое слово `ONLY`.

Вы также можете добавить после имени таблицы `*`, чтобы явно указать, что должны включаться и дочерние таблицы:

```
SELECT name, elevation
    FROM cities*
    WHERE elevation > 500;
```

Указывать `*` не обязательно, так как теперь это поведение всегда подразумевается по умолчанию. Однако такая запись всё ещё поддерживается для совместимости со старыми версиями, где поведение по умолчанию могло быть изменено.

В некоторых ситуациях бывает необходимо узнать, из какой таблицы выбрана конкретная строка. Для этого вы можете воспользоваться системным столбцом `tableoid`, присутствующим в каждой таблице:

```
SELECT c.tableoid, c.name, c.elevation
FROM cities c
WHERE c.elevation > 500;
```

этот запрос выдаст:

```
 tableoid |   name    | elevation
----------+-----------+-----------
   139793 | Las Vegas |      2174
   139793 | Mariposa  |      1953
   139798 | Madison   |       845
```

(Если вы попытаетесь выполнить его у себя, скорее всего вы получите другие значения OID.) Собственно имена таблиц вы можете получить, обратившись к `pg_class`:

```
SELECT p.relname, c.name, c.elevation
```

```
FROM cities c, pg_class p
WHERE c.elevation > 500 AND c.tableoid = p.oid;
```

в результате вы получите:

```
 relname  |   name    | elevation
----------+-----------+-----------
 cities   | Las Vegas |      2174
 cities   | Mariposa  |      1953
 capitals | Madison   |       845
```

Тот же эффект можно получить другим способом, используя альтернативный тип `regclass`; при этом OID таблицы выводится в символьном виде:

```
SELECT c.tableoid::regclass, c.name, c.elevation
FROM cities c
WHERE c.elevation > 500;
```

Механизм наследования не способен автоматически распределять данные команд `INSERT` или `COPY` по таблицам в иерархии наследования. Поэтому в нашем примере этот оператор `INSERT` не выполнится:

```
INSERT INTO cities (name, population, elevation, state)
VALUES ('Albany', NULL, NULL, 'NY');
```

Мы могли надеяться на то, что данные каким-то образом попадут в таблицу `capitals`, но этого не происходит: `INSERT` всегда вставляет данные непосредственно в указанную таблицу. В некоторых случаях добавляемые данные можно перенаправлять, используя правила (см. Главу 38). Однако в нашем случае это не поможет, так как таблица `cities` не содержит столбца `state` и команда будет отвергнута до применения правила.

Дочерние таблицы автоматически наследуют от родительской таблицы ограничения-проверки и ограничения NOT NULL (если только для них не задано явно `NO INHERIT`). Все остальные ограничения (уникальности, первичный ключ и внешние ключи) не наследуются.

Таблица может наследоваться от нескольких родительских таблиц, в этом случае она будет объединять в себе все столбцы этих таблиц, а также столбцы, описанные непосредственно в её определении. Если в определениях родительских и дочерней таблиц встретятся столбцы с одним именем, эти столбцы будут «объединены», так что в дочерней таблице окажется только один столбец. Чтобы такое объединение было возможно, столбцы должны иметь одинаковый тип данных, в противном случае произойдёт ошибка. Наследуемые ограничения-проверки и ограничения NOT NULL объединяются подобным образом. Так, например, объединяемый столбец получит свойство NOT NULL, если какое-либо из порождающих его определений имеет свойство NOT NULL. Ограничения-проверки объединяются, если они имеют одинаковые имена; но если их условия различаются, происходит ошибка.

Отношение наследования между таблицами обычно устанавливается при создании дочерней таблицы с использованием предложения `INHERITS` оператора CREATE TABLE. Другой способ добавить такое отношение для таблицы, определённой подходящим образом, — использовать `INHERIT` с оператором ALTER TABLE. Для этого будущая дочерняя таблица должна уже включать те же столбцы (с совпадающими именами и типами), что и родительская таблица. Также она должна включать аналогичные ограничения-проверки (с теми же именами и выражениями). Удалить отношение наследования можно с помощью указания `NO INHERIT` оператора `ALTER TABLE`. Динамическое добавление и удаление отношений наследования может быть полезно при реализации секционирования таблиц (см. Раздел 5.10).

Для создания таблицы, которая затем может стать наследником другой, удобно воспользоваться предложением `LIKE` оператора `CREATE TABLE`. Такая команда создаст новую таблицу с теми же столбцами, что имеются в исходной. Если в исходной таблицы определены ограничения `CHECK`, для создания полностью совместимой таблицы их тоже нужно скопировать, и это можно сделать, добавив к предложению `LIKE` параметр `INCLUDING CONSTRAINTS`.

Родительскую таблицу нельзя удалить, пока существуют унаследованные от неё. При этом в дочерних таблицах нельзя удалять или модифицировать столбцы или ограничения-проверки, унаследованные от родительских таблиц. Если вы хотите удалить таблицу вместе со всеми её потомками, это легко сделать, добавив в команду удаления родительской таблицы параметр CASCADE (см. Раздел 5.13).

При изменениях определений и ограничений столбцов команда ALTER TABLE распространяет эти изменения вниз в иерархии наследования. Однако удалить столбцы, унаследованные дочерними таблицами, можно только с помощью параметра CASCADE. При создании отношений наследования команда ALTER TABLE следует тем же правилам объединения дублирующихся столбцов, что и CREATE TABLE.

В запросах с наследуемыми таблицами проверка прав доступа выполняется только в родительской таблице. Так, например, наличие разрешения UPDATE для таблицы cities подразумевает право на изменение строк и в таблице capitals, когда это изменение осуществляется через cities. Тем самым поддерживается видимость того, что данные находятся (также) в родительской таблице. Но изменить данные непосредственно в таблице capitals нельзя без дополнительного разрешения. Это правило имеет два исключения — команды TRUNCATE и LOCK TABLE, при выполнении которых всегда проверяются разрешения и для дочерних таблиц (то есть прямое обращение к таблицам и косвенное, через родительскую, обрабатываются одинаково).

Подобным образом, политики защиты на уровне строк (см. Раздел 5.7) для родительской таблицы применяются к строкам, получаемым из дочерних таблиц при выполнении запроса с наследованием. Политики же дочерних таблиц, если они определены, действуют только когда такие таблицы явно задействуются в запросе; в этом случае все политики, связанные с родительскими таблицами, игнорируются.

Сторонние таблицы (см. Раздел 5.11) могут также входить в иерархию наследования как родительские или дочерние таблицы, так же, как и обычные. Если в иерархию наследования входит сторонняя таблица, все операции, не поддерживаемые ей, не будут поддерживаться иерархией в целом.

5.9.1. Ограничения

Заметьте, что не все SQL-команды могут работать с иерархиями наследования. Команды, выполняющие выборку данных, изменение данных или модификацию схемы (например SELECT, UPDATE, DELETE, большинство вариантов ALTER TABLE, но не INSERT и ALTER TABLE ... RENAME), обычно по умолчанию обрабатывают данные дочерних таблиц и могут исключать их, если поддерживают указание ONLY. Команды для обслуживания и настройки базы данных (например REINDEX и VACUUM) обычно работают только с отдельными физическими таблицами и не поддерживают рекурсивную обработку отношений наследования. Соответствующее поведение каждой команды описано в её справке (Команды SQL).

Возможности наследования серьёзно ограничены тем, что индексы (включая ограничения уникальности) и ограничения внешних ключей относятся только к отдельным таблицам, но не к их потомкам. Это касается обеих сторон ограничений внешних ключей. Таким образом, применительно к нашему примеру:

- Если мы объявим cities.name с ограничением UNIQUE или PRIMARY KEY, это не помешает добавить в таблицу capitals строки с названиями городов, уже существующими в таблице cities. И эти дублирующиеся строки по умолчанию будут выводиться в результате запросов к cities. На деле таблица capitals по умолчанию вообще не будет содержать ограничение уникальности, так что в ней могут оказаться несколько строк с одним названием. Хотя вы можете добавить в capitals соответствующее ограничение, но это не предотвратит дублирование при объединении с cities.
- Подобным образом, если мы укажем, что cities.name ссылается (REFERENCES) на какую-то другую таблицу, это ограничение не будет автоматически распространено на capitals. В этом случае решением может стать явное добавление такого же ограничения REFERENCES в таблицу capitals.

- Если вы сделаете, чтобы столбец другой таблицы ссылался на `cities(name)`, в этом столбце можно будет указывать только названия городов, но не столиц. В этом случае хорошего решения нет.

Некоторая функциональность, не реализованная для иерархий наследования, реализована для декларативного секционирования. Поэтому обязательно взвесьте все за и против, прежде чем применять в своих приложениях секционирование с использованием устаревшего наследования.

5.10. Секционирование таблиц

Postgres Pro поддерживает простое секционирование таблиц. В этом разделе описывается, как и почему бывает полезно применять секционирование при проектировании баз данных.

5.10.1. Обзор

Секционированием данных называется разбиение одной большой логической таблицы на несколько меньших физических секций. Секционирование может принести следующую пользу:

- В определённых ситуациях оно кардинально увеличивает быстродействие, особенно когда большой процент часто запрашиваемых строк таблицы относится к одной или лишь нескольким секциям. Секционирование может сыграть роль ведущих столбцов в индексах, что позволит уменьшить размер индекса и увеличит вероятность нахождения наиболее востребованных частей индексов в памяти.
- Когда в выборке или изменении данных задействована большая часть одной секции, последовательное сканирование этой секции может выполняться гораздо быстрее, чем случайный доступ по индексу к данным, разбросанным по всей таблице.
- Массовую загрузку и удаление данных можно осуществлять, добавляя и удаляя секции, если это было предусмотрено при проектировании секционированных таблиц. Операция `ALTER TABLE DETACH PARTITION` или удаление отдельной секции с помощью команды `DROP TABLE` выполняются гораздо быстрее, чем массовая обработка. Эти команды также полностью исключают накладные расходы, связанные с выполнением `VACUUM` после `DELETE`.
- Редко используемые данные можно перенести на более дешёвые и медленные носители.

Всё это обычно полезно только для очень больших таблиц. Какие именно таблицы выиграют от секционирования, зависит от конкретного приложения, хотя, как правило, это следует применять для таблиц, размер которых превышает объём ОЗУ сервера.

Postgres Pro предлагает поддержку следующих видов секционирования:

Секционирование по диапазонам

Таблица секционируется по «диапазонам», определённым по ключевому столбцу или набору столбцов, и не пересекающимся друг с другом. Например, можно секционировать данные по диапазонам дат или по диапазонам идентификаторов определённых бизнес-объектов.

Секционирование по списку

Таблица секционируется с помощью списка, явно указывающего, какие значения ключа должны относиться к каждой секции.

Если вашему приложению требуются другие формы секционирования, можно также прибегнуть к альтернативным реализациям, с использованием наследования и представлений с `UNION ALL`. Такие подходы дают гибкость, но не дают такого выигрыша в производительности, как встроенное декларативное секционирование.

5.10.2. Декларативное секционирование

Postgres Pro предоставляет возможность указать, как разбить таблицу на части, называемые секциями. Разделённая таким способом таблица называется *секционированной таблицей*. Указание секционирования состоит из определения *метода секционирования* и списка столбцов или выражений, которые будут составлять *ключ разбиения*.

Все строки, вставляемые в секционированную таблицу, будут распределяться по *секциям* в зависимости от значения ключа разбиения. Каждая секция содержит подмножество данных, ограниченное *границами секции*. В настоящее время в качестве методов секционирования поддерживается разбиение по диапазонам и по спискам, когда каждой секции назначается диапазон ключей или список ключей, соответственно.

Сами секции могут представлять собой секционируемые таблицы, благодаря применению так называемого *вложенного секционирования*. В секциях могут быть определены свои индексы, ограничения и значения по умолчанию, отличные от других секций. Индексы должны создаваться для каждой секции независимо. Подробнее о секционированных таблицах и секциях рассказывается в описании CREATE TABLE.

Преобразовать обычную таблицу в секционированную и наоборот нельзя. Однако в секционированную таблицу можно добавить в качестве секции обычную или секционированную таблицу с данными, а также можно удалить секцию из секционированной таблицы и превратить её в отдельную таблицу; обратитесь к описанию ALTER TABLE, чтобы узнать больше о подкомандах `ATTACH PARTITION` и `DETACH PARTITION`.

За кулисами отдельные секции связываются с секционируемой таблицей средствами наследования; однако с секционированными таблицами и секциями нельзя полноценно использовать функциональность наследования, рассматриваемую в предыдущем разделе. Например, секция не может иметь никаких других родителей, кроме секционированной таблицы, к которой она присоединена, так же как обычная таблица не может наследоваться от секционированной таблицы, чтобы последняя стала её родителем. Это значит, что секционированные таблицы и секции не участвуют в наследовании наряду с обычными таблицами. Так как иерархия наследования, включающая секционированную таблицу и её секции, остаётся иерархией наследования, на неё распространяются все обычные правила наследования, описанные в Раздел 5.9 с некоторыми исключениями, а именно:

- Ограничения `CHECK` вместе с `NOT NULL`, определённые в секционированной таблице, всегда наследуются всеми её секциями. Ограничения `CHECK` с характеристикой `NO INHERIT` в секционированных таблицах создавать нельзя.
- Использование указания `ONLY` при добавлении или удалении ограничения только в секционированной таблице поддерживается лишь когда в ней нет секций. Если секции существуют, при попытке использования `ONLY` возникнет ошибка, так как добавление или удаление ограничений только в секционированной таблице при наличии секций не поддерживается. С другой стороны, ограничения можно добавлять или удалять непосредственно в секциях, если они отсутствуют в родительской таблице. Так как секционированная таблица непосредственно не содержит данные, при попытке использовать `TRUNCATE ONLY` в секционированной таблице всегда возвращается ошибка.
- В секциях не может быть столбцов, отсутствующих в родительской таблице. Такие столбцы невозможно определить ни при создании секций командой `CREATE TABLE`, ни путём добавления в существующие секции с использованием `ALTER TABLE`. Таблицы могут быть подключены в качестве секций командой `ALTER TABLE ... ATTACH PARTITION`, только если их столбцы в точности соответствуют родительской таблице, включая столбцы `oid`.
- Ограничение `NOT NULL` для столбца в секции нельзя удалить, если это ограничение существует в родительской таблице.

Секции также могут быть сторонними таблицами (см. CREATE FOREIGN TABLE), хотя при этом действуют дополнительные ограничения по сравнению с обычными таблицами. Например, данные, вставляемые в секционированную таблицу, не перенаправляются в секцию, представляющую собой стороннюю таблицу.

5.10.2.1. Пример

Предположим, что мы создаём базу данных для большой компании, торгующей мороженым. Компания учитывает максимальную температуру и продажи мороженого каждый день в разрезе регионов. По сути нам нужна следующая таблица:

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
);
```

Мы знаем, что большинство запросов будут работать только с данными за последнюю неделю, месяц или квартал, так как в основном эта таблица нужна для формирования текущих отчётов для руководства. Чтобы сократить объём хранящихся старых данных, мы решили оставлять данные только за 3 последних года. Ненужные данные мы будем удалять в начале каждого месяца. В этой ситуации мы можем использовать секционирование для удовлетворения всех наших требований к таблице показателей.

Чтобы использовать декларативное секционирование в этом случае, выполните следующее:

1. Создайте таблицу `measurement` как секционированную таблицу с предложением `PARTITION BY`, указав метод разбиения (в нашем случае `RANGE`) и список столбцов, которые будут образовывать ключ разбиения.

   ```
   CREATE TABLE measurement (
       city_id         int not null,
       logdate         date not null,
       peaktemp        int,
       unitsales       int
   ) PARTITION BY RANGE (logdate);
   ```

   При разбиении по диапазонам в качестве ключа разбиения при желании можно использовать набор из нескольких столбцов. Конечно, при этом скорее всего увеличится количество секций, и каждая из них будет меньше. И напротив, использование меньшего числа столбцов может привести к менее дробному критерию разбиения с меньшим числом секций. Запрос, обращающийся к секционированной таблице, будет сканировать меньше секций, если в условии поиска фигурируют некоторые или все эти столбцы. Например, в таблице, секционируемой по диапазонам, в качестве ключа разбиения можно выбрать столбцы `lastname` и `firstname` (в таком порядке).
2. Создайте секции. В определении каждой секции должны задаваться границы, соответствующие методу и ключу разбиения родительской таблицы. Заметьте, что указание границ, при котором множество значений новой секции пересекается со множеством значений в одной или нескольких существующих секциях, будет ошибочным. При попытке добавления в родительскую таблицу данных, которые не соответствуют ни одной из существующей секций, произойдёт ошибка; соответствующий раздел нужно добавлять вручную.

   Секции, создаваемые таким образом, во всех отношениях являются обычными таблицами Postgres Pro (или, возможно, сторонними таблицами). В частности, для каждой секции можно независимо задать табличное пространство и параметры хранения.

   Для таблиц-секций нет необходимости определять ограничения с условиями, задающими границы значений. Нужные ограничения секций выводятся неявно из определения границ секции, когда требуется к ним обратиться.

   ```
   CREATE TABLE measurement_y2006m02 PARTITION OF measurement
       FOR VALUES FROM ('2006-02-01') TO ('2006-03-01');

   CREATE TABLE measurement_y2006m03 PARTITION OF measurement
       FOR VALUES FROM ('2006-03-01') TO ('2006-04-01');

   ...
   CREATE TABLE measurement_y2007m11 PARTITION OF measurement
       FOR VALUES FROM ('2007-11-01') TO ('2007-12-01');

   CREATE TABLE measurement_y2007m12 PARTITION OF measurement
   ```

```
    FOR VALUES FROM ('2007-12-01') TO ('2008-01-01')
    TABLESPACE fasttablespace;

CREATE TABLE measurement_y2008m01 PARTITION OF measurement
    FOR VALUES FROM ('2008-01-01') TO ('2008-02-01')
    WITH (parallel_workers = 4)
    TABLESPACE fasttablespace;
```

Для реализации вложенного секционирования укажите предложение `PARTITION BY` в командах, создающих отдельные секции, например:

```
CREATE TABLE measurement_y2006m02 PARTITION OF measurement
    FOR VALUES FROM ('2006-02-01') TO ('2006-03-01')
    PARTITION BY RANGE (peaktemp);
```

Когда будут созданы секции `measurement_y2006m02`, данные, добавляемые в `measurement` и попадающие в `measurement_y2006m02` (или данные, непосредственно добавляемые в `measurement_y2006m02`, с учётом соответствия ограничению секции) будут затем перенаправлены в одну из вложенных секций в зависимости от значения столбца `peaktemp`. Указанный ключ разбиения может пересекаться с ключом разбиения родителя, хотя определять границы вложенной секции нужно осмотрительно, чтобы множество данных, которое она принимает, входило во множество, допускаемое собственными границами секции; система не пытается контролировать это сама.

3. Создайте индекс по ключевому столбцу(ам), а также любые другие индексы, которые вы хотели бы иметь в каждой секции. (Индекс по ключу, строго говоря, не необходим, но в большинстве случаев он будет полезен. Если вы хотите, чтобы значения ключа были уникальны, вам следует также создать ограничения уникальности или первичного ключа для каждой секции.)

```
CREATE INDEX ON measurement_y2006m02 (logdate);
CREATE INDEX ON measurement_y2006m03 (logdate);
...
CREATE INDEX ON measurement_y2007m11 (logdate);
CREATE INDEX ON measurement_y2007m12 (logdate);
CREATE INDEX ON measurement_y2008m01 (logdate);
```

4. Убедитесь в том, что параметр конфигурации constraint_exclusion не выключен в `postgresql.conf`. Иначе запросы не будут оптимизироваться должным образом.

В данном примере нам потребуется создавать секцию каждый месяц, так что было бы разумно написать скрипт, который бы формировал требуемый код DDL автоматически.

5.10.2.2. Обслуживание секций

Обычно набор секций, образованный изначально при создании таблиц, не предполагается сохранять неизменным. Чаще наоборот, планируется удалять старые секции данных и периодически добавлять новые. Одно из наиболее важных преимуществ секционирования состоит именно в том, что оно позволяет практически моментально выполнять трудоёмкие операции, изменяя структуру секций, а не физически перемещая большие объёмы данных.

Самый лёгкий способ удалить старые данные — просто удалить секцию, ставшую ненужной:

```
DROP TABLE measurement_y2006m02;
```

Так можно удалить миллионы записей гораздо быстрее, чем удалять их по одной. Заметьте, однако, что приведённая выше команда требует установления блокировки `ACCESS EXCLUSIVE`.

Ещё один часто более предпочтительный вариант — убрать секцию из главной таблицы, но сохранить возможность обращаться к ней как к самостоятельной таблице:

```
ALTER TABLE measurement DETACH PARTITION measurement_y2006m02;
```

При этом можно будет продолжать работать с данными, пока таблица не будет удалена. Например, в этом состоянии очень кстати будет сделать резервную копию данных, используя `COPY`, pg_dump

или подобные средства. Возможно, эти данные также можно будет агрегировать, перевести в компактный формат, выполнить другую обработку или построить отчёты.

Аналогичным образом можно добавлять новую секцию с данными. Мы можем создать пустую секцию в главной таблице так же, как мы создавали секции в исходном состоянии до этого:

```
CREATE TABLE measurement_y2008m02 PARTITION OF measurement
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01')
    TABLESPACE fasttablespace;
```

А иногда удобнее создать новую таблицу вне структуры секций и сделать её полноценной секцией позже. При таком подходе данные можно будет загрузить, проверить и преобразовать до того, как они появятся в секционированной таблице:

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS)
  TABLESPACE fasttablespace;

ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );

\copy measurement_y2008m02 from 'measurement_y2008m02'
-- possibly some other data preparation work

ALTER TABLE measurement ATTACH PARTITION measurement_y2008m02
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01' );
```

Прежде чем выполнять команду `ATTACH PARTITION`, рекомендуется создать ограничение `CHECK` в присоединяемой таблице, соответствующее ожидаемому ограничению секции. Благодаря этому система сможет не сканировать таблицу для проверки неявного ограничения секции. Без этого ограничения `CHECK` таблицу нужно будет просканировать и убедиться в выполнении ограничения секции, удерживая блокировку `ACCESS EXCLUSIVE` в родительской таблице. После выполнения команды `ATTACH PARTITION` это ставшее ненужным ограничение `CHECK` при желании можно удалить.

5.10.2.3. Ограничения

С секционированными таблицами связаны следующие ограничения:

- Механизм автоматического создания соответствующих индексов во всех секциях отсутствует, поэтому индексы нужно добавлять в каждую секцию отдельно. Это также означает, что невозможно создать первичный ключ, ограничение уникальности или ограничение исключения, охватывающие все секции; такие ограничения возможны только в отдельных секциях.
- Так как в секционированных таблицах первичные ключи не поддерживаются, на секционированные таблицы не могут ссылаться внешние ключи, так же как и внешние ключи в сторонних таблицах не могут ссылаться на какие-либо другие таблицы.
- Использование предложения `ON CONFLICT` с секционированными таблицами вызовет ошибку, так как ограничения уникальности и ограничения исключения могут создаваться только в отдельных секциях. Обеспечение уникальности (или ограничения исключения) во всей иерархии наследования не поддерживается.
- Перенести строку из одной секции в другую командой `UPDATE` нельзя, так как новое значение строки не будет удовлетворять неявному ограничению исходной секции.
- Триггеры уровня строк при необходимости должны определяться в отдельных секциях, а не в секционированной таблице.
- Смешивание временных и постоянных отношений в одном дереве секционирования не допускается. Таким образом, если секционированная таблица постоянная, такими же должны

быть её секции; с временными таблицами аналогично. В случае с временными отношениями все таблицы дерева секционирования должны быть из одного сеанса.

5.10.3. Реализация с использованием наследования

Хотя встроенные средства декларативного секционирования полезны во многих часто возникающих ситуациях, бывают обстоятельства, требующие более гибкого подхода. В этом случае секционирование можно реализовать, применив механизм наследования таблиц, что даст ряд возможностей, неподдерживаемых при декларативном секционировании, например:

- С секционированием действует правило, что все секции должны иметь в точности тот же набор столбцов, что и родительская таблица, но собственно наследование таблиц допускает наличие в дочерних таблицах дополнительных столбцов, отсутствующих в родителе.
- Механизм наследования таблиц поддерживает множественное наследование.
- С декларативным секционированием поддерживается только разбиение по спискам и по диапазонам, тогда как с наследованием таблиц данные можно разделять по любому критерию, выбранному пользователем. (Однако заметьте, что если исключение по ограничению не позволяет эффективно отфильтровывать секции, производительность запросов будет очень низкой.)
- Для некоторых операций с декларативным секционированием требуется более сильная блокировка, чем с использованием наследования. Например, для добавления или удаления секций из секционированной таблицы требуется установить блокировку `ACCESS EXCLUSIVE` в родительской таблице, тогда как в случае с обычным наследованием достаточно блокировки `SHARE UPDATE EXCLUSIVE`.

5.10.3.1. Пример

Воспользуемся описанной ранее несекционированной таблицей `measurement`. Чтобы реализовать секционирование с использованием наследования, выполните следующие действия:

1. Создайте «главную» таблицу, от которой будут наследоваться все секции. Эта таблица не будет содержать данные. Не определяйте в ней никаких ограничений, если только вы не намерены создавать идентичные во всех секциях. Также не имеет смысла определять в ней какие-либо индексы или ограничения уникальности. В нашем примере главной таблицей будет `measurement` со своим изначальным определением.
2. Создайте несколько «дочерних» таблиц, унаследовав их все от главной. Обычно в таких таблицах не будет никаких дополнительных столбцов, кроме унаследованных. Как и с декларативным секционированием, эти секции во всех отношениях будут обычными таблицами Postgres Pro (или сторонними таблицами).

```
CREATE TABLE measurement_y2006m02 () INHERITS (measurement);
CREATE TABLE measurement_y2006m03 () INHERITS (measurement);
...
CREATE TABLE measurement_y2007m11 () INHERITS (measurement);
CREATE TABLE measurement_y2007m12 () INHERITS (measurement);
CREATE TABLE measurement_y2008m01 () INHERITS (measurement);
```

3. Добавьте в таблицы-секции неперекрывающиеся ограничения, определяющие допустимые значения ключей для каждой секции.

   Типичные примеры таких ограничений:

   ```
   CHECK ( x = 1 )
   CHECK ( county IN ( 'Oxfordshire', 'Buckinghamshire', 'Warwickshire' ))
   CHECK ( outletID >= 100 AND outletID < 200 )
   ```

   Убедитесь в том, что ограничения не пересекаются, то есть никакие значения ключа не относятся сразу к нескольким секциям. Например, часто допускают такую ошибку в определении диапазонов:

   ```
   CHECK ( outletID BETWEEN 100 AND 200 )
   CHECK ( outletID BETWEEN 200 AND 300 )
   ```

Это не будет работать, так как неясно, к какой секции должно относиться значение 200.

Секции лучше будет создать следующим образом:

```
CREATE TABLE measurement_y2006m02 (
    CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2006m03 (
    CHECK ( logdate >= DATE '2006-03-01' AND logdate < DATE '2006-04-01' )
) INHERITS (measurement);

...
CREATE TABLE measurement_y2007m11 (
    CHECK ( logdate >= DATE '2007-11-01' AND logdate < DATE '2007-12-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2007m12 (
    CHECK ( logdate >= DATE '2007-12-01' AND logdate < DATE '2008-01-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2008m01 (
    CHECK ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
) INHERITS (measurement);
```

4. Для каждой секции создайте индекс по ключевому столбцу(ам), а также любые другие индексы по своему усмотрению.

```
CREATE INDEX measurement_y2006m02_logdate ON measurement_y2006m02 (logdate);
CREATE INDEX measurement_y2006m03_logdate ON measurement_y2006m03 (logdate);
CREATE INDEX measurement_y2007m11_logdate ON measurement_y2007m11 (logdate);
CREATE INDEX measurement_y2007m12_logdate ON measurement_y2007m12 (logdate);
CREATE INDEX measurement_y2008m01_logdate ON measurement_y2008m01 (logdate);
```

5. Мы хотим, чтобы наше приложение могло сказать `INSERT INTO measurement ...` и данные оказались в соответствующей таблице-секции. Мы можем добиться этого, добавив подходящую триггерную функцию в главную таблицу. Если данные всегда будут добавляться только в последнюю секцию, нам будет достаточно очень простой функции:

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

Создав эту функцию, мы создадим вызывающий её триггер:

```
CREATE TRIGGER insert_measurement_trigger
    BEFORE INSERT ON measurement
    FOR EACH ROW EXECUTE PROCEDURE measurement_insert_trigger();
```

Мы должны менять определение триггерной функции каждый месяц, чтобы она всегда указывала на текущую секцию. Определение самого триггера, однако, менять не требуется.

Но мы можем также сделать, чтобы сервер автоматически находил секцию, в которую нужно направить добавляемую строку. Для этого нам потребуется более сложная триггерная функция:

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
```

```
    IF ( NEW.logdate >= DATE '2006-02-01' AND
         NEW.logdate < DATE '2006-03-01' ) THEN
        INSERT INTO measurement_y2006m02 VALUES (NEW.*);
    ELSIF ( NEW.logdate >= DATE '2006-03-01' AND
            NEW.logdate < DATE '2006-04-01' ) THEN
        INSERT INTO measurement_y2006m03 VALUES (NEW.*);
    ...
    ELSIF ( NEW.logdate >= DATE '2008-01-01' AND
            NEW.logdate < DATE '2008-02-01' ) THEN
        INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION
  'Date out of range.  Fix the measurement_insert_trigger() function!';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

Определение триггера остаётся прежним. Заметьте, что все условия `IF` должны в точности отражать ограничения `CHECK` соответствующих секций.

Хотя эта функция сложнее, чем вариант с одним текущим месяцем, её не придётся так часто модифицировать, так как ветви условий можно добавить заранее.

> **Примечание**
>
> На практике будет лучше сначала проверять условие для последней секции, если строки чаще добавляются в эту секцию. Для простоты же мы расположили проверки триггера в том же порядке, что и в других фрагментах кода для этого примера.

Другой способ перенаправления добавляемых строк в соответствующую секцию можно реализовать, определив для главной таблицы не триггер, а правила. Например:

```
CREATE RULE measurement_insert_y2006m02 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
DO INSTEAD
    INSERT INTO measurement_y2006m02 VALUES (NEW.*);
...
CREATE RULE measurement_insert_y2008m01 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
DO INSTEAD
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
```

С правилами связано гораздо больше накладных расходов, чем с триггером, но они относятся к запросу в целом, а не к каждой строке. Поэтому этот способ может быть более выигрышным при массовом добавлении данных. Однако в большинстве случаев триггеры будут работать быстрее.

Учтите, что команда `COPY` игнорирует правила. Если вы хотите вставить данные с помощью `COPY`, вам придётся копировать их сразу в нужную секцию, а не в главную таблицу. С другой стороны, `COPY` не отменяет триггеры, так что с триггерами вы сможете использовать её обычным образом.

Ещё один недостаток подхода с правилами связан с невозможностью выдать ошибку, если добавляемая строка не подпадает ни под одно из правил; в этом случае данные просто попадут в главную таблицу.

6. Убедитесь в том, что параметр конфигурации constraint_exclusion не выключен в `postgresql.conf`. Иначе запросы не будут оптимизироваться должным образом.

Как уже можно понять, для реализации сложной схемы разбиения может потребоваться DDL-код значительного объёма. В данном примере нам потребуется создавать секцию каждый месяц, так что было бы разумно написать скрипт, который бы формировал требуемый код DDL автоматически.

5.10.3.2. Обслуживание секций

Чтобы быстро удалить старые данные, просто удалите ставшую ненужной секцию:

```
DROP TABLE measurement_y2006m02;
```

Чтобы удалить секцию из секционированной таблицы, но сохранить к ней доступ как к самостоятельной таблице:

```
ALTER TABLE measurement_y2006m02 NO INHERIT measurement;
```

Чтобы добавить новую секцию для новых данных, создайте пустую секцию так же, как до этого создавали начальные секции:

```
CREATE TABLE measurement_y2008m02 (
    CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' )
) INHERITS (measurement);
```

Можно также создать новую таблицу вне структуры секций и сделать её полноценной секцией после того, как данные будут в неё загружены, проверены и при необходимости преобразованы.

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );
\copy measurement_y2008m02 from 'measurement_y2008m02'
-- возможна дополнительная подготовка данных
ALTER TABLE measurement_y2008m02 INHERIT measurement;
```

5.10.3.3. Ограничения

С секционированными таблицами, реализованными через наследование, связаны следующие ограничения:

- Система не может проверить автоматически, являются ли все ограничения `CHECK` взаимно исключающими. Поэтому безопаснее будет написать и отладить код для формирования секций и создания и/или изменения связанных объектов, чем делать это вручную.
- Показанные здесь схемы подразумевают, что ключевой столбец(ы) секции в строке никогда не меняется, или меняется не настолько, чтобы строку потребовалось перенести в другую секцию. Если же попытаться выполнить такой оператор `UPDATE`, произойдёт ошибка из-за нарушения ограничения `CHECK`. Если вам нужно обработать и такие случаи, вы можете установить подходящие триггеры на обновление в таблицы-секции, но это ещё больше усложнит управление всей конструкцией.
- Если вы выполняете команды `VACUUM` или `ANALYZE` вручную, не забывайте, что их нужно запускать для каждой секции в отдельности. Команда

  ```
  ANALYZE measurement;
  ```

  обработает только главную таблицу.
- Операторы `INSERT` с предложениями `ON CONFLICT` скорее всего не будут работать ожидаемым образом, так как действие `ON CONFLICT` предпринимается только в случае нарушений уникальности в указанном целевом отношении, а не его дочерних отношениях.
- Для направления строк в нужные секции потребуются триггеры или правила, если только приложение не знает непосредственно о схеме секционирования. Разработать триггеры

может быть довольно сложно, и они будут работать гораздо медленнее, чем внутреннее распределение кортежей при декларативном секционировании.

5.10.4. Секционирование и исключение по ограничению

Исключение по ограничению — это приём оптимизации запросов, который ускоряет работу с секционированными таблицами, определёнными по вышеописанной схеме (и для декларативного секционирования, и для секционирования с использованием наследования). Например:

```
SET constraint_exclusion = on;
SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
```

Без исключения по ограничению для данного запроса пришлось бы просканировать все секции таблицы `measurement`. Если же исключение по ограничению включено, планировщик рассмотрит ограничение каждой секции с целью определить, что данная секция не может содержать строки, удовлетворяющие условию запроса `WHERE`. Если планировщик придёт к такому выводу, он исключит эту секцию из плана запроса.

Чтобы увидеть, как меняется план при изменении параметра `constraint_exclusion`, вы можете воспользоваться командой `EXPLAIN`. Типичный неоптимизированный план для такой конфигурации таблицы будет выглядеть так:

```
SET constraint_exclusion = off;
EXPLAIN SELECT count(*) FROM measurement
  WHERE logdate >= DATE '2008-01-01';

                                QUERY PLAN
------------------------------------------------------------------------------
 Aggregate  (cost=158.66..158.68 rows=1 width=0)
   ->  Append  (cost=0.00..151.88 rows=2715 width=0)
         ->  Seq Scan on measurement  (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2006m02 measurement
                     (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2006m03 measurement
                     (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2008-01-01'::date)
...
         ->  Seq Scan on measurement_y2007m12 measurement
                     (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2008m01 measurement
                     (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2008-01-01'::date)
```

В некоторых или всех секциях может применяться не полное последовательное сканирование, а сканирование по индексу, но основная идея примера в том, что для удовлетворения запроса не нужно сканировать старые секции. И когда мы включаем исключение по ограничению, мы получаем значительно более эффективный план, дающий тот же результат:

```
SET constraint_exclusion = on;
EXPLAIN SELECT count(*) FROM measurement
  WHERE logdate >= DATE '2008-01-01';

                                QUERY PLAN
------------------------------------------------------------------------------
 Aggregate  (cost=63.47..63.48 rows=1 width=0)
   ->  Append  (cost=0.00..60.75 rows=1086 width=0)
         ->  Seq Scan on measurement  (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2008-01-01'::date)
```

```
->  Seq Scan on measurement_y2008m01 measurement
              (cost=0.00..30.38 rows=543 width=0)
      Filter: (logdate >= '2008-01-01'::date)
```

Заметьте, что механизм исключения по ограничению учитывает только ограничения `CHECK`, но не наличие индексов. Поэтому определять индексы для столбцов ключа не обязательно. Нужно ли создавать индекс для данной секции, зависит от того, какая часть секции будет обрабатываться при выполнении большинства запросов. Если это небольшая часть, индекс может быть полезен, в противном случае он не нужен.

По умолчанию параметр constraint_exclusion имеет значение не `on` и не `off`, а промежуточное (и рекомендуемое) значение `partition`, при котором этот приём будет применяться только к запросам, где предположительно будут задействованы секционированные таблицы. Значение `on` обязывает планировщик просматривать ограничения `CHECK` во всех запросах, даже в самых простых, где исключение по ограничению не будет иметь смысла.

Относительно исключения по ограничению, применяемого и при наследовании, и при секционировании таблиц, необходимо учитывать следующее:

- Исключение по ограничению работает только когда предложение `WHERE` в запросе содержит константы (или получаемые извне параметры). Например, сравнение с функцией переменной природы, такой как `CURRENT_TIMESTAMP`, нельзя оптимизировать, так как планировщик не может знать, в какую секцию попадёт значение функции во время выполнения.
- Ограничения секций должны быть простыми, иначе планировщик не сможет вычислить, какие секции не нужно обрабатывать. Используйте простые условия на равенство при секционировании по списку или простые проверки интервала при секционировании по диапазонам, как показано в предыдущих примерах. Рекомендуется использовать в секционирующих ограничениях только сравнения столбцов разбиения с константами с использованием операторов, индексируемых в B-дереве (это применимо и к секционированным таблицам, так как в ключ разбиения могут входить только столбцы, индексируемые в B-дереве). (Это не проблема при декларативном секционировании, так как автоматически генерируемые ограничения достаточно просты для понимания планировщика.)
- При анализе для исключения по ограничению исследуются все ограничения всех секций главной таблицы, поэтому при большом количестве секций время планирования запросов может значительно увеличиться. Описанные выше подходы работают хорошо, пока количество секций не превышает примерно ста, но не пытайтесь применять их с тысячами секций.

5.10.5. Рекомендации по декларативному секционированию

К секционированию таблицы следует подходить продуманно, так как неудачное решение может отрицательно повлиять на скорость планирования и выполнения запросов.

Одним из самых важных факторов является выбор столбца или столбцов, по которым будут секционироваться ваши данные. Часто оптимальным будет секционирование по столбцу или набору столбцов, которые практически всегда присутствуют в предложении `WHERE` в запросах, обращающихся к секционируемой таблице. Элементы предложения `WHERE`, совместимые с ключом секционирования и соответствующие ему, могут применяться для устранения ненужных для выполнения запроса секций. Также при планировании секционирования следует учитывать, как будут удаляться данные. Секцию целиком можно отсоединить очень быстро, поэтому может иметь смысл разработать стратегию секционирования так, чтобы массово удаляемые данные оказывались в одной секции.

Также важно правильно выбрать число секций, на которые будет разбиваться таблица. Если их будет недостаточно много, индексы останутся большими, не улучшится и локальность данных, вследствие чего процент попаданий в кеш окажется низким. Однако и при слишком большом количестве секций возможны проблемы. С большим количеством секций увеличивается время

планирования и потребление памяти как при планировании, так и при выполнении запросов. Выбирая стратегию секционирования таблицы, также важно учитывать, какие изменения могут произойти в будущем. Например, если вы решите создавать отдельные секции для каждого клиента, и в данный момент у вас всего несколько больших клиентов, подумайте, что будет, если через несколько лет у вас будет много мелких клиентов. В этом случае может быть лучше произвести секционирование по хешу (`HASH`) и выбрать разумное количество секций, по которым равномерно распределятся клиенты, но не создавать секции по диапазонам (`RANGE`) в надежде, что количество клиентов не увеличится до такой степени, что секционирование данных окажется непрактичным.

Вложенное секционирование позволяет дополнительно разделить те секции, которые предположительно окажутся больше других секций. Однако если этой возможностью злоупотреблять, может образоваться большое количество секций, порождающее те же проблемы, что описаны в предыдущем абзаце.

Также важно учитывать издержки секционирования, которые отражаются на планировании и выполнении запросов. Планировщик запросов обычно довольно неплохо справляется с иерархиями, включающими до нескольких сотен секций. Однако по мере увеличения числа секций будет увеличиваться и время планирования запросов, и объём потребляемой памяти. В наибольшей степени это касается команд `UPDATE` и `DELETE`. Наличие большого количества секций нежелательно ещё и потому, что потребление памяти сервером может значительно возрастать с течением времени, особенно когда множество сеансов обращаются к множеству секций. Это объясняется тем, что в локальную память каждого сеанса, который обращается к секциям, должны быть загружены метаданные всех этих секций.

С нагрузкой, присущей информационным хранилищам, может иметь смысл создавать больше секций, чем с нагрузкой OLTP. Как правило, в информационных хранилищах время планирования запроса второстепенно, так как гораздо больше времени тратится на выполнение запроса. Однако при любом типе нагрузки важно принимать правильные решения на ранней стадии реализации, так как процесс переразбиения таблиц большого объёма может оказаться чрезвычайно длительным. Для оптимизации стратегии секционирования часто бывает полезно предварительно эмулировать ожидаемую нагрузку. Но ни в коем случае нельзя полагать, что чем больше секций, тем лучше, равно как и наоборот.

5.11. Сторонние данные

Postgres Pro частично реализует спецификацию SQL/MED, позволяя вам обращаться к данным, находящимся снаружи, используя обычные SQL-запросы. Такие данные называются *сторонними*.

Сторонние данные доступны в PostgreSQL через *обёртку сторонних данных*. Обёртка сторонних данных — это библиотека, взаимодействующая с внешним источником данных и скрывающая в себе внутренние особенности подключения и получения данных. Несколько готовых обёрток предоставляются в виде модулей `contrib` ; см. Приложение F. Также вы можете найти другие обёртки, выпускаемые как дополнительные продукты. Если ни одна из существующих обёрток вас не устраивает, вы можете написать свою собственную (см. Главу 52).

Чтобы обратиться к сторонним данным, вы должны создать объект *сторонний сервер*, в котором настраивается подключение к внешнему источнику данных, определяются параметры соответствующей обёртки сторонних данных. Затем вы должны создать одну или несколько *сторонних таблиц*, определив тем самым структуру внешних данных. Сторонние таблицы можно использовать в запросах так же, как и обычные, но их данные не хранятся на сервере Postgres Pro. При каждом запросе Postgres Pro обращается к обёртке сторонних данных, которая, в свою очередь, получает данные из внешнего источника или передаёт их ему (в случае команд INSERT или UPDATE).

При обращении к внешним данным удалённый источник может потребовать аутентификации клиента. Соответствующие учётные данные можно предоставить с помощью *сопоставлений пользователей*, позволяющих определить в частности имена и пароли, в зависимости от текущей роли пользователя Postgres Pro.

Дополнительную информацию вы найдёте в CREATE FOREIGN DATA WRAPPER, CREATE SERVER, CREATE USER MAPPING, CREATE FOREIGN TABLE и IMPORT FOREIGN SCHEMA.

5.12. Другие объекты баз данных

Таблицы — центральные объекты в структуре реляционной базы данных, так как они содержат ваши данные. Но это не единственные объекты, которые могут в ней существовать. Помимо них вы можете создавать и использовать объекты и других типов, призванные сделать управление данными эффективнее и удобнее. Они не обсуждаются в этой главе, но мы просто перечислим некоторые из них, чтобы вы знали об их существовании:

- Представления
- Функции и операторы
- Типы данных и домены
- Триггеры и правила перезаписи

Подробнее соответствующие темы освещаются в Части V.

5.13. Отслеживание зависимостей

Когда вы создаёте сложные структуры баз данных, включающие множество таблиц с внешними ключами, представлениями, триггерами, функциями и т. п., вы неявно создаёте сеть зависимостей между объектами. Например, таблица с ограничением внешнего ключа зависит от таблицы, на которую она ссылается.

Для сохранения целостности структуры всей базы данных, Postgres Pro не позволяет удалять объекты, от которых зависят другие. Например, попытка удалить таблицу products (мы рассматривали её в Подразделе 5.3.5), от которой зависит таблица orders, приведёт к ошибке примерно такого содержания:

```
DROP TABLE products;

ОШИБКА:  удалить объект "таблица products" нельзя, так как от него зависят другие
ПОДРОБНОСТИ:  ограничение orders_product_no_fkey в отношении "таблица orders" зависит
 от объекта "таблица products"
ПОДСКАЗКА:  Для удаления зависимых объектов используйте DROP ... CASCADE.
```

Сообщение об ошибке даёт полезную подсказку: если вы не хотите заниматься ликвидацией зависимостей по отдельности, можно выполнить:

```
DROP TABLE products CASCADE;
```

и все зависимые объекты, а также объекты, зависящие от них, будут удалены рекурсивно. В этом случае таблица orders останется, а удалено будет только её ограничение внешнего ключа. Удаление не распространится на другие объекты, так как ни один объект не зависит от этого ограничения. (Если вы хотите проверить, что произойдёт при выполнении DROP ... CASCADE, запустите DROP без CASCADE и прочитайте ПОДРОБНОСТИ (DETAIL).)

Почти все команды DROP в Postgres Pro поддерживают указание CASCADE. Конечно, вид возможных зависимостей зависит от типа объекта. Вы также можете написать RESTRICT вместо CASCADE, чтобы включить поведение по умолчанию, когда объект можно удалить, только если от него не зависят никакие другие.

> **Примечание**
>
> Стандарт SQL требует явного указания RESTRICT или CASCADE в команде DROP. Но это требование на самом деле не выполняется ни в одной СУБД, при этом одни системы по умолчанию подразумевают RESTRICT, а другие — CASCADE.

Если в команде DROP перечисляются несколько объектов, CASCADE требуется указывать, только когда есть зависимости вне заданной группы. Например, в команде DROP TABLE tab1, tab2 при наличии внешнего ключа, ссылающегося на tab1 из tab2, можно не указывать CASCADE, чтобы она выполнилась успешно.

Для пользовательских функций Postgres Pro отслеживает зависимости, связанные с внешне видимыми свойствами функции, такими как типы аргументов и результата, но *не* зависимости, которые могут быть выявлены только при анализе тела функции. В качестве примера рассмотрите следующий сценарий:

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow',
                             'green', 'blue', 'purple');

CREATE TABLE my_colors (color rainbow, note text);

CREATE FUNCTION get_color_note (rainbow) RETURNS text AS
  'SELECT note FROM my_colors WHERE color = $1'
  LANGUAGE SQL;
```

(Описание функций языка SQL можно найти в Разделе 35.4.) Postgres Pro будет понимать, что функция get_color_note зависит от типа rainbow: при удалении типа будет принудительно удалена функция, так как тип её аргумента оказывается неопределённым. Но Postgres Pro не будет учитывать зависимость get_color_note от таблицы my_colors и не удалит функцию при удалении таблицы. Но у этого подхода есть не только минус, но и плюс. В случае отсутствия таблицы эта функция останется рабочей в некотором смысле: хотя при попытке выполнить её возникнет ошибка, но при создании новой таблицы с тем же именем функция снова будет работать.

Глава 6. Модификация данных

В предыдущей главе мы обсуждали, как создавать таблицы и другие структуры для хранения данных. Теперь пришло время заполнить таблицы данными. В этой главе мы расскажем, как добавлять, изменять и удалять данные из таблиц. А из следующей главы вы наконец узнаете, как извлекать нужные вам данные из базы данных.

6.1. Добавление данных

Сразу после создания таблицы она не содержит никаких данных. Поэтому, чтобы она была полезна, в неё прежде всего нужно добавить данные. По сути данные добавляются в таблицу по одной строке. И хотя вы конечно можете добавить в таблицу несколько строк, добавить в неё меньше, чем строку, невозможно. Даже если вы указываете значения только некоторых столбцов, создаётся полная строка.

Чтобы создать строку, вы будете использовать команду INSERT. В этой команде необходимо указать имя таблицы и значения столбцов. Например, рассмотрим таблицу товаров из Главы 5:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

Добавить в неё строку можно было бы так:

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

Значения данных перечисляются в порядке столбцов в таблице и разделяются запятыми. Обычно в качестве значений указываются константы, но это могут быть и скалярные выражения.

Показанная выше запись имеет один недостаток — вам необходимо знать порядок столбцов в таблице. Чтобы избежать этого, можно перечислить столбцы явно. Например, следующие две команды дадут тот же результат, что и показанная выше:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

Многие считают, что лучше всегда явно указывать имена столбцов.

Если значения определяются не для всех столбцов, лишние столбцы можно опустить. В таком случае эти столбцы получат значения по умолчанию. Например:

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');
INSERT INTO products VALUES (1, 'Cheese');
```

Вторая форма является расширением Postgres Pro. Она заполняет столбцы слева по числу переданных значений, а все остальные столбцы принимают значения по умолчанию.

Для ясности можно также явно указать значения по умолчанию для отдельных столбцов или всей строки:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', DEFAULT);
INSERT INTO products DEFAULT VALUES;
```

Одна команда может вставить сразу несколько строк:

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);
```

Также возможно вставить результат запроса (который может не содержать строк либо содержать одну или несколько):

```
INSERT INTO products (product_no, name, price)
  SELECT product_no, name, price FROM new_products
    WHERE release_date = 'today';
```

Это позволяет использовать все возможности механизма запросов SQL (см. Главу 7) для вычисления вставляемых строк.

> **Подсказка**
>
> Когда нужно добавить сразу множество строк, возможно будет лучше использовать команду COPY. Она не такая гибкая, как INSERT, но гораздо эффективнее. Дополнительно об ускорении массовой загрузки данных можно узнать в Разделе 14.4.

6.2. Изменение данных

Модификация данных, уже сохранённых в БД, называется изменением. Изменить можно все строки таблицы, либо подмножество всех строк, либо только избранные строки. Каждый столбец при этом можно изменять независимо от других.

Для изменения данных в существующих строках используется команда UPDATE. Ей требуется следующая информация:

1. Имя таблицы и изменяемого столбца
2. Новое значение столбца
3. Критерий отбора изменяемых строк

Если вы помните, в Главе 5 говорилось, что в SQL в принципе нет уникального идентификатора строк. Таким образом, не всегда возможно явно указать на строку, которую требуется изменить. Поэтому необходимо указать условия, каким должны соответствовать требуемая строка. Только если в таблице есть первичный ключ (вне зависимости от того, объявляли вы его или нет), можно однозначно адресовать отдельные строки, определив условие по первичному ключу. Этим пользуются графические инструменты для работы с базой данных, дающие возможность редактировать данные по строкам.

Например, следующая команда увеличивает цену всех товаров, имевших до этого цену 5, до 10:

```
UPDATE products SET price = 10 WHERE price = 5;
```

В результате может измениться ноль, одна или множество строк. И если этому запросу не будет удовлетворять ни одна строка, это не будет ошибкой.

Давайте рассмотрим эту команду подробнее. Она начинается с ключевого слова `UPDATE`, за которым идёт имя таблицы. Как обычно, имя таблицы может быть записано в полной форме, в противном случае она будет найдена по пути. Затем идёт ключевое слово `SET`, за которым следует имя столбца, знак равенства и новое значение столбца. Этим значением может быть любое скалярное выражение, а не только константа. Например, если вы захотите поднять цену всех товаров на 10%, это можно сделать так:

```
UPDATE products SET price = price * 1.10;
```

Как видно из этого примера, выражение нового значения может ссылаться на существующие значения столбцов в строке. Мы также опустили в нём предложение `WHERE`. Это означает, что будут изменены все строки в таблице. Если же это предложение присутствует, изменяются только строки, которые соответствуют условию `WHERE`. Заметьте, что хотя знак равенства в предложении `SET` обозначает операцию присваивания, а такой же знак в предложении `WHERE` используется для сравнения, это не приводит к неоднозначности. И конечно, в условии `WHERE` не обязательно должна быть проверка равенства, а могут применяться и другие операторы (см. Главу 9). Необходимо только, чтобы это выражение возвращало логический результат.

В команде `UPDATE` можно изменить значения сразу нескольких столбцов, перечислив их в предложении `SET`. Например:

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

6.3. Удаление данных

Мы рассказали о том, как добавлять данные в таблицы и как изменять их. Теперь вам осталось узнать, как удалить данные, которые оказались не нужны. Так же, как добавлять данные можно только целыми строками, удалять их можно только по строкам. В предыдущем разделе мы отметили, что в SQL нет возможности напрямую адресовать отдельные строки, так что удалить избранные строки можно, только сформулировав для них подходящие условия. Но если в таблице есть первичный ключ, с его помощью можно однозначно выделить определённую строку. При этом можно так же удалить группы строк, соответствующие условию, либо сразу все строки таблицы.

Для удаления строк используется команда DELETE; её синтаксис очень похож на синтаксис команды UPDATE. Например, удалить все строки из таблицы с товарами, имеющими цену 10, можно так:

```
DELETE FROM products WHERE price = 10;
```

Если вы напишете просто:

```
DELETE FROM products;
```

будут удалены все строки таблицы! Будьте осторожны!

6.4. Возврат данных из изменённых строк

Иногда бывает полезно получать данные из модифицируемых строк в процессе их обработки. Это возможно с использованием предложения RETURNING, которое можно задать для команд INSERT, UPDATE и DELETE. Применение RETURNING позволяет обойтись без дополнительного запроса к базе для сбора данных и это особенно ценно, когда как-то иначе трудно получить изменённые строки надёжным образом.

В предложении RETURNING допускается то же содержимое, что и в выходном списке команды SELECT (см. Раздел 7.3). Оно может содержать имена столбцов целевой таблицы команды или значения выражений с этими столбцами. Также часто применяется краткая запись RETURNING *, выбирающая все столбцы целевой таблицы по порядку.

В команде INSERT данные, выдаваемые в RETURNING, образуются из строки в том виде, в каком она была вставлена. Это не очень полезно при простом добавлении, так как в результате будут получены те же данные, что были переданы клиентом. Но это может быть очень удобно при использовании вычисляемых значений по умолчанию. Например, если в таблице есть столбец serial, в котором генерируются уникальные идентификаторы, команда RETURNING может возвратить идентификатор, назначенный новой строке:

```
CREATE TABLE users (firstname text, lastname text, id serial primary key);

INSERT INTO users (firstname, lastname) VALUES ('Joe', 'Cool') RETURNING id;
```

Предложение RETURNING также очень полезно с INSERT ... SELECT.

В команде UPDATE данные, выдаваемые в RETURNING, образуются новым содержимым изменённой строки. Например:

```
UPDATE products SET price = price * 1.10
  WHERE price <= 99.99
  RETURNING name, price AS new_price;
```

В команде DELETE данные, выдаваемые в RETURNING, образуются содержимым удалённой строки. Например:

```
DELETE FROM products
  WHERE obsoletion_date = 'today'
```

```
  RETURNING *;
```

Если для целевой таблицы заданы триггеры (см. Главу 36), в `RETURNING` выдаются данные из строки, изменённой триггерами. Таким образом, `RETURNING` часто применяется и для того, чтобы проверить содержимое столбцов, изменяемых триггерами.

Глава 7. Запросы

В предыдущих главах рассказывалось, как создать таблицы, как заполнить их данными и как изменить эти данные. Теперь мы наконец обсудим, как получить данные из базы данных.

7.1. Обзор

Процесс или команда получения данных из базы данных называется *запросом*. В SQL запросы формулируются с помощью команды SELECT. В общем виде команда SELECT записывается так:

```
[WITH запросы_with] SELECT список_выборки FROM табличное_выражение
 [определение_сортировки]
```

В следующих разделах подробно описываются список выборки, табличное выражение и определение сортировки. Запросы WITH являются расширенной возможностью PostgreSQL и будут рассмотрены в последнюю очередь.

Простой запрос выглядит так:

```
SELECT * FROM table1;
```

Если предположить, что в базе данных есть таблица table1, эта команда получит все строки с содержимым всех столбцов из table1. (Метод выдачи результата определяет клиентское приложение. Например, программа psql выведет на экране ASCII-таблицу, хотя клиентские библиотеки позволяют извлекать отдельные значения из результата запроса.) Здесь список выборки задан как *, это означает, что запрос должен вернуть все столбцы табличного выражения. В списке выборки можно также указать подмножество доступных столбцов или составить выражения с этими столбцами. Например, если в table1 есть столбцы a, b и c (и возможно, другие), вы можете выполнить следующий запрос:

```
SELECT a, b + c FROM table1;
```

(в предположении, что столбцы b и c имеют числовой тип данных). Подробнее это описано в Разделе 7.3.

FROM table1 — это простейший тип табличного выражения, в котором просто читается одна таблица. Вообще табличные выражения могут быть сложными конструкциями из базовых таблиц, соединений и подзапросов. А можно и вовсе опустить табличное выражение и использовать команду SELECT как калькулятор:

```
SELECT 3 * 4;
```

В этом может быть больше смысла, когда выражения в списке выборки возвращают меняющиеся результаты. Например, можно вызвать функцию так:

```
SELECT random();
```

7.2. Табличные выражения

Табличное выражение вычисляет таблицу. Это выражение содержит предложение FROM, за которым могут следовать предложения WHERE, GROUP BY и HAVING. Тривиальные табличные выражения просто ссылаются на физическую таблицу, её называют также базовой, но в более сложных выражениях такие таблицы можно преобразовывать и комбинировать самыми разными способами.

Необязательные предложения WHERE, GROUP BY и HAVING в табличном выражении определяют последовательность преобразований, осуществляемых с данными таблицы, полученной в предложении FROM. В результате этих преобразований образуется виртуальная таблица, строки которой передаются списку выборки, вычисляющему выходные строки запроса.

7.2.1. Предложение FROM

«Предложение FROM» образует таблицу из одной или нескольких ссылок на таблицы, разделённых запятыми.

```
FROM табличная_ссылка [, табличная_ссылка [, ...]]
```

Здесь табличной ссылкой может быть имя таблицы (возможно, с именем схемы), производная таблица, например подзапрос, соединение таблиц или сложная комбинация этих вариантов. Если в предложении FROM перечисляются несколько ссылок, для них применяется перекрёстное соединение (то есть декартово произведение их строк; см. ниже). Список FROM преобразуется в промежуточную виртуальную таблицу, которая может пройти через преобразования WHERE, GROUP BY и HAVING, и в итоге определит результат табличного выражения.

Когда в табличной ссылке указывается таблица, являющаяся родительской в иерархии наследования, в результате будут получены строки не только этой таблицы, но и всех её дочерних таблиц. Чтобы выбрать строки только одной родительской таблицы, перед её именем нужно добавить ключевое слово ONLY. Учтите, что при этом будут получены только столбцы указанной таблицы — дополнительные столбцы дочерних таблиц не попадут в результат.

Если же вы не добавляете ONLY перед именем таблицы, вы можете дописать после него *, тем самым указав, что должны обрабатываться и все дочерние таблицы. Практических причин использовать этот синтаксис больше нет, так как поиск в дочерних таблицах теперь производится по умолчанию. Однако эта запись поддерживается для совместимости со старыми версиями.

7.2.1.1. Соединённые таблицы

Соединённая таблица — это таблица, полученная из двух других (реальных или производных от них) таблиц в соответствии с правилами соединения конкретного типа. Общий синтаксис описания соединённой таблицы:

```
T1 тип_соединения T2 [ условие_соединения ]
```

Соединения любых типов могут вкладываются друг в друга или объединяться: и *T1*, и *T2* могут быть результатами соединения. Для однозначного определения порядка соединений предложения JOIN можно заключать в скобки. Если скобки отсутствуют, предложения JOIN обрабатываются слева направо.

Типы соединений

Перекрёстное соединение

```
T1 CROSS JOIN T2
```

Соединённую таблицу образуют все возможные сочетания строк из *T1* и *T2* (т. е. их декартово произведение), а набор её столбцов объединяет в себе столбцы *T1* со следующими за ними столбцами *T2*. Если таблицы содержат N и M строк, соединённая таблица будет содержать N * M строк.

FROM *T1* CROSS JOIN *T2* равнозначно FROM *T1* INNER JOIN *T2* ON TRUE (см. ниже). Эта запись также равнозначна FROM *T1*, *T2*.

> **Примечание**
>
> Последняя запись не полностью эквивалентна первым при указании более чем двух таблиц, так как JOIN связывает таблицы сильнее, чем запятая. Например, FROM *T1* CROSS JOIN *T2* INNER JOIN *T3* ON *условие* не равнозначно FROM *T1*, *T2* INNER JOIN *T3* ON *условие*, так как *условие* может ссылаться на *T1* в первом случае, но не во втором.

Соединения с сопоставлениями строк

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
  ON логическое_выражение
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
  USING ( список столбцов соединения )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

Слова INNER и OUTER необязательны во всех формах. По умолчанию подразумевается INNER (внутреннее соединение), а при указании LEFT, RIGHT и FULL — внешнее соединение.

Условие соединения указывается в предложении ON или USING, либо неявно задаётся ключевым словом NATURAL. Это условие определяет, какие строки двух исходных таблиц считаются «соответствующими» друг другу (это подробно рассматривается ниже).

Возможные типы соединений с сопоставлениями строк:

INNER JOIN

Для каждой строки R1 из T1 в результирующей таблице содержится строка для каждой строки в T2, удовлетворяющей условию соединения с R1.

LEFT OUTER JOIN

Сначала выполняется внутреннее соединение (INNER JOIN). Затем в результат добавляются все строки из T1, которым не соответствуют никакие строки в T2, а вместо значений столбцов T2 вставляются NULL. Таким образом, в результирующей таблице всегда будет минимум одна строка для каждой строки из T1.

RIGHT OUTER JOIN

Сначала выполняется внутреннее соединение (INNER JOIN). Затем в результат добавляются все строки из T2, которым не соответствуют никакие строки в T1, а вместо значений столбцов T1 вставляются NULL. Это соединение является обратным к левому (LEFT JOIN): в результирующей таблице всегда будет минимум одна строка для каждой строки из T2.

FULL OUTER JOIN

Сначала выполняется внутреннее соединение. Затем в результат добавляются все строки из T1, которым не соответствуют никакие строки в T2, а вместо значений столбцов T2 вставляются NULL. И наконец, в результат включаются все строки из T2, которым не соответствуют никакие строки в T1, а вместо значений столбцов T1 вставляются NULL.

Предложение ON определяет наиболее общую форму условия соединения: в нём указываются выражения логического типа, подобные тем, что используются в предложении WHERE. Пара строк из *T1* и *T2* соответствуют друг другу, если выражение ON возвращает для них true.

USING — это сокращённая запись условия, полезная в ситуации, когда с обеих сторон соединения столбцы имеют одинаковые имена. Она принимает список общих имён столбцов через запятую и формирует условие соединения с равенством этих столбцов. Например, запись соединения *T1* и *T2* с USING (a, b) формирует условие ON *T1*.a = *T2*.a AND *T1*.b = *T2*.b.

Более того, при выводе JOIN USING исключаются избыточные столбцы: оба сопоставленных столбца выводить не нужно, так как они содержат одинаковые значения. Тогда как JOIN ON выдаёт все столбцы из *T1*, а за ними все столбцы из *T2*, JOIN USING выводит один столбец для каждой пары (в указанном порядке), за ними все оставшиеся столбцы из *T1* и, наконец, все оставшиеся столбцы *T2*.

Наконец, NATURAL — сокращённая форма USING: она образует список USING из всех имён столбцов, существующих в обеих входных таблицах. Как и с USING, эти столбцы оказываются в выходной таблице в единственном экземпляре. Если столбцов с одинаковыми именами не находится, NATURAL JOIN действует как JOIN ... ON TRUE и выдаёт декартово произведение строк.

Примечание

Предложение USING разумно защищено от изменений в соединяемых отношениях, так как оно связывает только явно перечисленные столбцы. NATURAL считается более рискованным, так как при любом изменении схемы в одном или другом отношении, когда

появляются столбцы с совпадающими именами, при соединении будут связываться и эти новые столбцы.

Для наглядности предположим, что у нас есть таблицы `t1`:

```
 num | name
-----+------
   1 | a
   2 | b
   3 | c
```

и `t2`:

```
 num | value
-----+-------
   1 | xxx
   3 | yyy
   5 | zzz
```

С ними для разных типов соединений мы получим следующие результаты:

```
=> SELECT * FROM t1 CROSS JOIN t2;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   1 | a    |   3 | yyy
   1 | a    |   5 | zzz
   2 | b    |   1 | xxx
   2 | b    |   3 | yyy
   2 | b    |   5 | zzz
   3 | c    |   1 | xxx
   3 | c    |   3 | yyy
   3 | c    |   5 | zzz
(9 rows)

=> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
(2 rows)

=> SELECT * FROM t1 INNER JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 NATURAL INNER JOIN t2;
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
```

```
   2 | b    |     |
   3 | c    |   3 | yyy
(3 rows)

=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   2 | b    |
   3 | c    | yyy
(3 rows)

=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
     |      |   5 | zzz
(3 rows)

=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
     |      |   5 | zzz
(4 rows)
```

Условие соединения в предложении `ON` может также содержать выражения, не связанные непосредственно с соединением. Это может быть полезно в некоторых запросах, но не следует использовать это необдуманно. Рассмотрите следующий запрос:

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |     |
(3 rows)
```

Заметьте, что если поместить ограничение в предложение `WHERE`, вы получите другой результат:

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num WHERE t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
(1 row)
```

Это связано с тем, что ограничение, помещённое в предложение `ON`, обрабатывается *до* операции соединения, тогда как ограничение в `WHERE` — *после*. Это не имеет значения при внутренних соединениях, но важно при внешних.

7.2.1.2. Псевдонимы таблиц и столбцов

Таблицам и ссылкам на сложные таблицы в запросе можно дать временное имя, по которому к ним можно будет обращаться в рамках запроса. Такое имя называется *псевдонимом таблицы*.

Определить псевдоним таблицы можно, написав

```
FROM табличная_ссылка AS псевдоним
```

или

```
FROM табличная_ссылка псевдоним
```

Ключевое слово `AS` является необязательным. Вместо *псевдоним* здесь может быть любой идентификатор.

Псевдонимы часто применяются для назначения коротких идентификаторов длинным именам таблиц с целью улучшения читаемости запросов. Например:

```
SELECT * FROM "очень_длинное_имя_таблицы" s JOIN "другое_длинное_имя" a
  ON s.id = a.num;
```

Псевдоним становится новым именем таблицы в рамках текущего запроса, т. е. после назначения псевдонима использовать исходное имя таблицы в другом месте запроса нельзя. Таким образом, следующий запрос недопустим:

```
SELECT * FROM my_table AS m WHERE my_table.a > 5;    -- неправильно
```

Хотя в основном псевдонимы используются для удобства, они бывают необходимы, когда таблица соединяется сама с собой, например:

```
SELECT * FROM people AS mother JOIN people AS child
  ON mother.id = child.mother_id;
```

Кроме того, псевдонимы обязательно нужно назначать подзапросам (см. Подраздел 7.2.1.3).

В случае неоднозначности определения псевдонимов можно использовать скобки. В следующем примере первый оператор назначает псевдоним `b` второму экземпляру `my_table`, а второй оператор назначает псевдоним результату соединения:

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

В другой форме назначения псевдонима временные имена даются не только таблицам, но и её столбцам:

```
FROM табличная_ссылка [AS] псевдоним ( столбец1 [, столбец2 [, ...]] )
```

Если псевдонимов столбцов оказывается меньше, чем фактически столбцов в таблице, остальные столбцы сохраняют свои исходные имена. Эта запись особенно полезна для замкнутых соединений или подзапросов.

Когда псевдоним применяется к результату `JOIN`, он скрывает оригинальные имена таблиц внутри `JOIN`. Например, это допустимый SQL-запрос:

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

а запрос:

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

ошибочный, так как псевдоним таблицы `a` не виден снаружи определения псевдонима `c`.

7.2.1.3. Подзапросы

Подзапросы, образующие таблицы, должны заключаться в скобки и им *обязательно* должны назначаться псевдонимы (как описано в Подразделе 7.2.1.2). Например:

```
FROM (SELECT * FROM table1) AS псевдоним
```

Этот пример равносилен записи `FROM table1 AS псевдоним`. Более интересные ситуации, которые нельзя свести к простому соединению, возникают, когда в подзапросе используются агрегирующие функции или группировка.

Подзапросом может также быть список `VALUES`:

```
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))
     AS names(first, last)
```

Такому подзапросу тоже требуется псевдоним. Назначать псевдонимы столбцам списка VALUES не требуется, но вообще это хороший приём. Подробнее это описано в Разделе 7.7.

7.2.1.4. Табличные функции

Табличные функции — это функции, выдающие набор строк, содержащих либо базовые типы данных (скалярных типов), либо составные типы (табличные строки). Они применяются в запросах как таблицы, представления или подзапросы в предложении FROM. Столбцы, возвращённые табличными функциями, можно включить в выражения SELECT, JOIN или WHERE так же, как столбцы таблиц, представлений или подзапросов.

Табличные функции можно также скомбинировать, используя запись ROWS FROM. Результаты функций будут возвращены в параллельных столбцах; число строк в этом случае будет наибольшим из результатов всех функций, а результаты функций с меньшим количеством строк будут дополнены значениями NULL.

```
вызов_функции [WITH ORDINALITY] [[AS] псевдоним_таблицы [(псевдоним_столбца [, ...])]]
ROWS FROM( вызов_функции [, ...] ) [WITH ORDINALITY] [[AS] псевдоним_таблицы
 [(псевдоним_столбца [, ...])]]
```

Если указано предложение WITH ORDINALITY, к столбцам результатов функций будет добавлен ещё один, с типом bigint. В этом столбце нумеруются строки результирующего набора, начиная с 1. (Это обобщение стандартного SQL-синтаксиса UNNEST ... WITH ORDINALITY.) По умолчанию, этот столбец называется ordinality, но ему можно присвоить и другое имя с помощью указания AS.

Специальную табличную функцию UNNEST можно вызвать с любым числом параметров-массивов, а возвращает она соответствующее число столбцов, как если бы UNNEST (Раздел 9.18) вызывалась для каждого параметра в отдельности, а результаты объединялись с помощью конструкции ROWS FROM.

```
UNNEST( выражение_массива [, ...] ) [WITH ORDINALITY] [[AS] псевдоним_таблицы
 [(псевдоним_столбца [, ...])]]
```

Если *псевдоним_таблицы* не указан, в качестве имени таблицы используется имя функции; в случае с конструкцией ROWS FROM() — имя первой функции.

Если псевдонимы столбцов не указаны, то для функции, возвращающей базовый тип данных, именем столбца будет имя функции. Для функций, возвращающих составной тип, имена результирующих столбцов определяются индивидуальными атрибутами типа.

Несколько примеров:

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

SELECT * FROM foo
    WHERE foosubid IN (
                        SELECT foosubid
                        FROM getfoo(foo.fooid) z
                        WHERE z.fooid = foo.fooid
                      );

CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);
```

```
SELECT * FROM vw_getfoo;
```

В некоторых случаях бывает удобно определить табличную функцию, возвращающую различные наборы столбцов при разных вариантах вызова. Это можно сделать, объявив функцию, не имеющую выходных параметров (OUT) и возвращающую псевдотип record. Используя такую функцию, ожидаемую структуру строк нужно описать в самом запросе, чтобы система знала, как разобрать запрос и составить его план. Записывается это так:

```
вызов_функции [AS] псевдоним (определение_столбца [, ...])
вызов_функции AS [псевдоним] (определение_столбца [, ...])
ROWS FROM( ... вызов_функции AS (определение_столбца [, ...]) [, ...] )
```

Без ROWS FROM() список *определения_столбцов* заменяет список псевдонимов, который можно также добавить в предложении FROM; имена в определениях столбцов служат псевдонимами. С ROWS FROM() список *определения_столбцов* можно добавить к каждой функции отдельно, либо в случае с одной функцией и без предложения WITH ORDINALITY, список *определения_столбцов* можно записать вместо списка с псевдонимами столбцов после ROWS FROM().

Взгляните на этот пример:

```
SELECT *
    FROM dblink('dbname=mydb', 'SELECT proname, prosrc FROM pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

Здесь функция dblink (из модуля dblink) выполняет удалённый запрос. Она объявлена как функция, возвращающая тип record, так как он подойдёт для запроса любого типа. В этом случае фактический набор столбцов функции необходимо описать в вызывающем её запросе, чтобы анализатор запроса знал, например, как преобразовать *.

В этом примере используется конструкция ROWS FROM:

```
SELECT *
FROM ROWS FROM
    (
        json_to_recordset('[{"a":40,"b":"foo"},{"a":"100","b":"bar"}]')
            AS (a INTEGER, b TEXT),
        generate_series(1, 3)
    ) AS x (p, q, s)
ORDER BY p;

  p  |  q  | s
-----+-----+---
  40 | foo | 1
 100 | bar | 2
     |     | 3
```

Она объединяет результаты двух функций в одном отношении FROM. В данном случае json_to_recordset() должна выдавать два столбца, первый integer и второй text, а результат generate_series() используется непосредственно. Предложение ORDER BY упорядочивает значения первого столбца как целочисленные.

7.2.1.5. Подзапросы LATERAL

Перед подзапросами в предложении FROM можно добавить ключевое слово LATERAL. Это позволит ссылаться в них на столбцы предшествующих элементов списка FROM. (Без LATERAL каждый подзапрос выполняется независимо и поэтому не может обращаться к другим элементам FROM.)

Перед табличными функциями в предложении FROM также можно указать LATERAL, но для них это ключевое слово необязательно; в аргументах функций в любом случае можно обращаться к столбцам в предыдущих элементах FROM.

Элемент LATERAL может находиться на верхнем уровне списка FROM или в дереве JOIN. В последнем случае он может также ссылаться на любые элементы в левой части JOIN, справа от которого он находится.

Когда элемент FROM содержит ссылки LATERAL, запрос выполняется следующим образом: сначала для строки элемента FROM с целевыми столбцами, или набора строк из нескольких элементов FROM, содержащих целевые столбцы, вычисляется элемент LATERAL со значениями этих столбцов. Затем результирующие строки обычным образом соединяются со строками, из которых они были вычислены. Эта процедура повторяется для всех строк исходных таблиц.

LATERAL можно использовать так:

```
SELECT * FROM foo, LATERAL (SELECT * FROM bar WHERE bar.id = foo.bar_id) ss;
```

Здесь это не очень полезно, так как тот же результат можно получить более простым и привычным способом:

```
SELECT * FROM foo, bar WHERE bar.id = foo.bar_id;
```

Применять LATERAL имеет смысл в основном, когда для вычисления соединяемых строк необходимо обратиться к столбцам других таблиц. В частности, это полезно, когда нужно передать значение функции, возвращающей набор данных. Например, если предположить, что vertices(polygon) возвращает набор вершин многоугольника, близкие вершины многоугольников из таблицы polygons можно получить так:

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1, polygons p2,
     LATERAL vertices(p1.poly) v1,
     LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

Этот запрос можно записать и так:

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1 CROSS JOIN LATERAL vertices(p1.poly) v1,
     polygons p2 CROSS JOIN LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

или переформулировать другими способами. (Как уже упоминалось, в данном примере ключевое слово LATERAL не требуется, но мы добавили его для ясности.)

Особенно полезно бывает использовать LEFT JOIN с подзапросом LATERAL, чтобы исходные строки оказывались в результате, даже если подзапрос LATERAL не возвращает строк. Например, если функция get_product_names() выдаёт названия продуктов, выпущенных определённым производителем, но о продукции некоторых производителей информации нет, мы можем найти, каких именно, примерно так:

```
SELECT m.name
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true
WHERE pname IS NULL;
```

7.2.2. Предложение WHERE

«Предложение WHERE» записывается так:

```
WHERE условие_ограничения
```

где *условие_ограничения* — любое выражение значения (см. Раздел 4.2), выдающее результат типа boolean.

После обработки предложения FROM каждая строка полученной виртуальной таблицы проходит проверку по условию ограничения. Если результат условия равен true, эта строка остаётся в выходной таблице, а иначе (если результат равен false или NULL) отбрасывается. В условии

ограничения, как правило, задействуется минимум один столбец из таблицы, полученной на выходе FROM. Хотя строго говоря, это не требуется, но в противном случае предложение WHERE будет бессмысленным.

> **Примечание**
>
> Условие для внутреннего соединения можно записать как в предложении WHERE, так и в предложении JOIN. Например, это выражение:
>
> ```
> FROM a, b WHERE a.id = b.id AND b.val > 5
> ```
>
> равнозначно этому:
>
> ```
> FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
> ```
>
> и возможно, даже этому:
>
> ```
> FROM a NATURAL JOIN b WHERE b.val > 5
> ```
>
> Какой вариант выбрать, в основном дело вкуса и стиля. Вариант с JOIN внутри предложения FROM, возможно, не лучший с точки зрения совместимости с другими СУБД, хотя он и описан в стандарте SQL. Но для внешних соединений других вариантов нет: их можно записывать только во FROM. Предложения ON и USING во внешних соединениях *не* равнозначны условию WHERE, так как они могут добавлять строки (для входных строк без соответствия), а также удалять их из конечного результата.

Несколько примеров запросов с WHERE:

```
SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN
  (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

fdt — название таблицы, порождённой в предложении FROM. Строки, которые не соответствуют условию WHERE, исключаются из fdt. Обратите внимание, как в качестве выражений значения используются скалярные подзапросы. Как и любые другие запросы, подзапросы могут содержать сложные табличные выражения. Заметьте также, что fdt используется в подзапросах. Дополнение имени c1 в виде fdt.c1 необходимо только, если в порождённой таблице в подзапросе также оказывается столбец c1. Полное имя придаёт ясность даже там, где без него можно обойтись. Этот пример показывает, как область именования столбцов внешнего запроса распространяется на все вложенные в него внутренние запросы.

7.2.3. Предложения GROUP BY и HAVING

Строки порождённой входной таблицы, прошедшие фильтр WHERE, можно сгруппировать с помощью предложения GROUP BY, а затем оставить в результате только нужные группы строк, используя предложение HAVING.

```
SELECT список_выборки
    FROM ...
    [WHERE ...]
    GROUP BY группирующий_столбец [, группирующий_столбец]...
```

«Предложение GROUP BY» группирует строки таблицы, объединяя их в одну группу при совпадении значений во всех перечисленных столбцах. Порядок, в котором указаны столбцы, не имеет значения. В результате наборы строк с одинаковыми значениями преобразуются в отдельные строки, представляющие все строки группы. Это может быть полезно для устранения избыточности выходных данных и/или для вычисления агрегатных функций, применённых к этим группам. Например:

```
=> SELECT * FROM test1;
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM test1 GROUP BY x;
 x
---
 a
 b
 c
(3 rows)
```

Во втором запросе мы не могли написать `SELECT * FROM test1 GROUP BY x`, так как для столбца `y` нет единого значения, связанного с каждой группой. Однако столбцы, по которым выполняется группировка, можно использовать в списке выборки, так как они имеют единственное значение в каждой группе.

Вообще говоря, в группированной таблице столбцы, не включённые в список `GROUP BY`, можно использовать только в агрегатных выражениях. Пример такого агрегатного выражения:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;
 x | sum
---+-----
 a |   4
 b |   5
 c |   2
(3 rows)
```

Здесь `sum` — агрегатная функция, вычисляющая единственное значение для всей группы. Подробную информацию о существующих агрегатных функциях можно найти в Разделе 9.20.

> **Подсказка**
>
> Группировка без агрегатных выражений по сути выдаёт набор различающихся значений столбцов. Этот же результат можно получить с помощью предложения `DISTINCT` (см. Подраздел 7.3.3).

Взгляните на следующий пример: в нём вычисляется общая сумма продаж по каждому продукту (а не общая сумма по всем продуктам):

```
SELECT product_id, p.name, (sum(s.units) * p.price) AS sales
    FROM products p LEFT JOIN sales s USING (product_id)
    GROUP BY product_id, p.name, p.price;
```

В этом примере столбцы `product_id`, `p.name` и `p.price` должны присутствовать в списке `GROUP BY`, так как они используются в списке выборки. Столбец `s.units` может отсутствовать в списке `GROUP BY`, так как он используется только в агрегатном выражении (`sum(...)`), вычисляющем

сумму продаж. Для каждого продукта этот запрос возвращает строку с итоговой суммой по всем продажам данного продукта.

Если бы в таблице products по столбцу product_id был создан первичный ключ, тогда в данном примере было бы достаточно сгруппировать строки по product_id, так как название и цена продукта *функционально зависят* от кода продукта и можно однозначно определить, какое название и цену возвращать для каждой группы по ID.

В стандарте SQL GROUP BY может группировать только по столбцам исходной таблицы, но расширение Postgres Pro позволяет использовать в GROUP BY столбцы из списка выборки. Также возможна группировка по выражениям, а не просто именам столбцов.

Если таблица была сгруппирована с помощью GROUP BY, но интерес представляют только некоторые группы, отфильтровать их можно с помощью предложения HAVING, действующего подобно WHERE. Записывается это так:

```
SELECT список_выборки FROM ... [WHERE ...] GROUP BY ...
  HAVING логическое_выражение
```

В предложении HAVING могут использоваться и группирующие выражения, и выражения, не участвующие в группировке (в этом случае это должны быть агрегирующие функции).

Пример:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)

=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)
```

И ещё один более реалистичный пример:

```
SELECT product_id, p.name, (sum(s.units) * (p.price - p.cost)) AS profit
    FROM products p LEFT JOIN sales s USING (product_id)
    WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
    GROUP BY product_id, p.name, p.price, p.cost
    HAVING sum(p.price * s.units) > 5000;
```

В данном примере предложение WHERE выбирает строки по столбцу, не включённому в группировку (выражение истинно только для продаж за последние четыре недели), тогда как предложение HAVING отфильтровывает группы с общей суммой продаж больше 5000. Заметьте, что агрегатные выражения не обязательно должны быть одинаковыми во всех частях запроса.

Если в запросе есть вызовы агрегатных функций, но нет предложения GROUP BY, строки всё равно будут группироваться: в результате окажется одна строка группы (или возможно, ни одной строки, если эта строка будет отброшена предложением HAVING). Это справедливо и для запросов, которые содержат только предложение HAVING, но не содержат вызовы агрегатных функций и предложение GROUP BY.

7.2.4. GROUPING SETS, CUBE и ROLLUP

Более сложные, чем описанные выше, операции группировки возможны с концепцией *наборов группирования*. Данные, выбранные предложениями FROM и WHERE, группируются отдельно для

каждого заданного набора группирования, затем для каждой группы вычисляются агрегатные функции как для простых предложений GROUP BY, и в конце возвращаются результаты. Например:

```
=> SELECT * FROM items_sold;
 brand | size | sales
-------+------+-------
 Foo   | L    |  10
 Foo   | M    |  20
 Bar   | M    |  15
 Bar   | L    |  5
(4 rows)

=> SELECT brand, size, sum(sales) FROM items_sold GROUP BY GROUPING SETS ((brand),
 (size), ());
 brand | size | sum
-------+------+-----
 Foo   |      |  30
 Bar   |      |  20
       | L    |  15
       | M    |  35
       |      |  50
(5 rows)
```

В каждом внутреннем списке GROUPING SETS могут задаваться ноль или более столбцов или выражений, которые воспринимаются так же, как если бы они были непосредственно записаны в предложении GROUP BY. Пустой набор группировки означает, что все строки сводятся к одной группе (которая выводится, даже если входных строк нет), как описано выше для агрегатных функций без предложения GROUP BY.

Ссылки на группирующие столбцы или выражения заменяются в результирующих строках значениями NULL для тех группирующих наборов, в которых эти столбцы отсутствуют. Чтобы можно было понять, результатом какого группирования стала конкретная выходная строка, предназначена функция, описанная в Таблице 9.56.

Для указания двух распространённых видов наборов группирования предусмотрена краткая запись. Предложение формы

```
ROLLUP ( e1, e2, e3, ... )
```

представляет заданный список выражений и всех префиксов списка, включая пустой список; то есть оно равнозначно записи

```
GROUPING SETS (
    ( e1, e2, e3, ... ),
    ...
    ( e1, e2 ),
    ( e1 ),
    ( )
)
```

Оно часто применяется для анализа иерархических данных, например, для суммирования зарплаты по отделам, подразделениям и компании в целом.

Предложение формы

```
CUBE ( e1, e2, ... )
```

представляет заданный список и все его возможные подмножества (степень множества). Таким образом, запись

```
CUBE ( a, b, c )
```

равнозначна

```
GROUPING SETS (
    ( a, b, c ),
    ( a, b    ),
    ( a,    c ),
    ( a       ),
    (    b, c ),
    (    b    ),
    (       c ),
    (         )
)
```

Элементами предложений `CUBE` и `ROLLUP` могут быть либо отдельные выражения, либо вложенные списки элементов в скобках. Вложенные списки обрабатываются как атомарные единицы, с которыми формируются отдельные наборы группирования. Например:

```
CUBE ( (a, b), (c, d) )
```

равнозначно

```
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b       ),
    (       c, d ),
    (            )
)
```

и

```
ROLLUP ( a, (b, c), d )
```

равнозначно

```
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b, c    ),
    ( a          ),
    (            )
)
```

Конструкции `CUBE` и `ROLLUP` могут применяться либо непосредственно в предложении `GROUP BY`, либо вкладываться внутрь предложения `GROUPING SETS`. Если одно предложение `GROUPING SETS` вкладывается внутрь другого, результат будет таким же, как если бы все элементы внутреннего предложения были записаны непосредственно во внешнем.

Если в одном предложении `GROUP BY` задаётся несколько элементов группирования, окончательный список наборов группирования образуется как прямое произведение этих элементов. Например:

```
GROUP BY a, CUBE (b, c), GROUPING SETS ((d), (e))
```

равнозначно

```
GROUP BY GROUPING SETS (
    (a, b, c, d), (a, b, c, e),
    (a, b, d),    (a, b, e),
    (a, c, d),    (a, c, e),
    (a, d),       (a, e)
)
```

Примечание

Конструкция `(a, b)` обычно воспринимается в выражениях как конструктор строки. Однако в предложении `GROUP BY` на верхнем уровне выражений запись `(a, b)` воспринимается

как список выражений, как описано выше. Если вам по какой-либо причине *нужен* именно конструктор строки в выражении группирования, используйте запись `ROW(a, b)`.

7.2.5. Обработка оконных функций

Если запрос содержит оконные функции (см. Раздел 3.5, Раздел 9.21 и Подраздел 4.2.8), эти функции вычисляются после каждой группировки, агрегатных выражений и фильтрации `HAVING`. Другими словами, если в запросе есть агрегатные функции, предложения `GROUP BY` или `HAVING`, оконные функции видят не исходные строки, полученные из `FROM`/`WHERE`, а сгруппированные.

Когда используются несколько оконных функций, все оконные функции, имеющие в своих определениях синтаксически равнозначные предложения `PARTITION BY` и `ORDER BY`, гарантированно обрабатывают данные за один проход. Таким образом, они увидят один порядок сортировки, даже если `ORDER BY` не определяет порядок однозначно. Однако относительно функций с разными формулировками `PARTITION BY` и `ORDER BY` никаких гарантий не даётся. (В таких случаях между проходами вычислений оконных функций обычно требуется дополнительный этап сортировки и эта сортировка может не сохранять порядок строк, равнозначный с точки зрения `ORDER BY`.)

В настоящее время оконные функции всегда требуют предварительно отсортированных данных, так что результат запроса будет отсортирован согласно тому или иному предложению `PARTITION BY`/`ORDER BY` оконных функций. Однако полагаться на это не следует. Если вы хотите, чтобы результаты сортировались определённым образом, явно добавьте предложение `ORDER BY` на верхнем уровне запроса.

7.3. Списки выборки

Как говорилось в предыдущем разделе, табличное выражение в `SELECT` создаёт промежуточную виртуальную таблицу, возможно объединяя таблицы, представления, группируя и исключая лишние строки и т. д. Полученная таблица передаётся для обработки в *список выборки*. Этот список выбирает, какие *столбцы* промежуточной таблицы должны выводиться в результате и как именно.

7.3.1. Элементы списка выборки

Простейший список выборки образует элемент `*`, который выбирает все столбцы из полученного табличного выражения. Список выборки также может содержать список выражений значения через запятую (как определено в Разделе 4.2). Например, это может быть список имён столбцов:

```
SELECT a, b, c FROM ...
```

Имена столбцов `a`, `b` и `c` представляют либо фактические имена столбцов таблиц, перечисленных в предложении `FROM`, либо их псевдонимы, определённые как описано в Подразделе 7.2.1.2. Пространство имён в списке выборки то же, что и в предложении `WHERE`, если не используется группировка. В противном случае оно совпадает с пространством имён предложения `HAVING`.

Если столбец с заданным именем есть в нескольких таблицах, необходимо также указать имя таблицы, например так:

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

Обращаясь к нескольким таблицам, бывает удобно получить сразу все столбцы одной из таблиц:

```
SELECT tbl1.*, tbl2.a FROM ...
```

Подробнее запись *`имя_таблицы`*`.*` описывается в Подразделе 8.16.5.

Если в списке выборки используется обычное выражение значения, по сути при этом в возвращаемую таблицу добавляется новый виртуальный столбец. Выражение значения вычисляется один раз для каждой строки результата со значениями столбцов в данной строке. Хотя выражения в списке выборки не обязательно должны обращаться к столбцам табличного

выражения из предложения FROM; они могут содержать, например и простые арифметические выражения.

7.3.2. Метки столбцов

Элементам в списке выборки можно назначить имена для последующей обработки, например, для указания в предложении ORDER BY или для вывода в клиентском приложении. Например:

```
SELECT a AS value, b + c AS sum FROM ...
```

Если выходное имя столбца не определено (с помощью AS), система назначает имя сама. Для простых ссылок на столбцы этим именем становится имя целевого столбца, а для вызовов функций это имя функции. Для сложных выражений система генерирует некоторое подходящее имя.

Слово AS можно опустить, но только если имя нового столбца не является ключевым словом Postgres Pro (см. Приложение C). Во избежание случайного совпадения имени с ключевым словом это имя можно заключить в кавычки. Например, VALUE — ключевое слово, поэтому такой вариант не будет работать:

```
SELECT a value, b + c AS sum FROM ...
```

а такой будет:

```
SELECT a "value", b + c AS sum FROM ...
```

Для предотвращения конфликта с ключевыми словами, которые могут появиться в будущем, рекомендуется всегда писать AS или заключать метки выходных столбцов в кавычки.

> **Примечание**
>
> Именование выходных столбцов отличается от того, что происходит в предложении FROM (см. Подраздел 7.2.1.2). Один столбец можно переименовать дважды, но на выходе окажется имя, назначенное в списке выборки.

7.3.3. DISTINCT

После обработки списка выборки в результирующей таблице можно дополнительно исключить дублирующиеся строки. Для этого сразу после SELECT добавляется ключевое слово DISTINCT:

```
SELECT DISTINCT список_выборки ...
```

(Чтобы явно включить поведение по умолчанию, когда возвращаются все строки, вместо DISTINCT можно указать ключевое слово ALL.)

Две строки считаются разными, если они содержат различные значения минимум в одном столбце. При этом значения NULL полагаются равными.

Кроме того, можно явно определить, какие строки будут считаться различными, следующим образом:

```
SELECT DISTINCT ON (выражение [, выражение ...]) список_выборки ...
```

Здесь *выражение* — обычное выражение значения, вычисляемое для всех строк. Строки, для которых перечисленные выражения дают один результат, считаются дублирующимися и возвращается только первая строка из такого набора. Заметьте, что «первая строка» набора может быть любой, если только запрос не включает сортировку, гарантирующую однозначный порядок строк, поступающих в фильтр DISTINCT. (Обработка DISTINCT ON производится после сортировки ORDER BY.)

Предложение DISTINCT ON не описано в стандарте SQL и иногда его применение считается плохим стилем из-за возможной неопределённости в результатах. При разумном использовании GROUP BY и подзапросов во FROM можно обойтись без этой конструкции, но часто она бывает удобнее.

7.4. Сочетание запросов

Результаты двух запросов можно обработать, используя операции над множествами: объединение, пересечение и вычитание. Эти операции записываются соответственно так:

запрос1 UNION [ALL] *запрос2*
запрос1 INTERSECT [ALL] *запрос2*
запрос1 EXCEPT [ALL] *запрос2*

Здесь *запрос1* и *запрос2* — это запросы, в которых могут использоваться все возможности, рассмотренные до этого.

UNION по сути добавляет результаты второго запроса к результатам первого (хотя никакой порядок возвращаемых строк при этом не гарантируется). Более того, эта операция убирает дублирующиеся строки из результата так же, как это делает DISTINCT, если только не указано UNION ALL.

INTERSECT возвращает все строки, содержащиеся в результате и первого, и второго запроса. Дублирующиеся строки отфильтровываются, если не указано ALL.

EXCEPT возвращает все строки, которые есть в результате первого запроса, но отсутствуют в результате второго. (Иногда это называют *разницей* двух запросов.) И здесь дублирующиеся строки отфильтровываются, если не указано ALL.

Чтобы можно было вычислить объединение, пересечение или разницу результатов двух запросов, эти запросы должны быть «совместимыми для объединения», что означает, что они должны иметь одинаковое число столбцов и соответствующие столбцы должны быть совместимых типов, как описывается в Разделе 10.5.

Операции над множествами можно объединять, например

запрос1 UNION *запрос2* EXCEPT *запрос3*

что равнозначно

(*запрос1* UNION *запрос2*) EXCEPT *запрос3*

Как показано выше, вы можете использовать круглые скобки, чтобы управлять очередью обработки. Без круглых скобок UNION и EXCEPT объединяются слева направо, но оператор INTERSECT имеет больший приоритет, чем первые два. Таким образом,

запрос1 UNION *запрос2* INTERSECT *запрос3*

означает

запрос1 UNION (*запрос2* INTERSECT *запрос3*)

Вы также можете заключить отдельный *запрос* в круглые скобки. Это важно, если в *запросе* необходимо использовать любое из предложений, обсуждаемых в следующих разделах, например LIMIT. Без круглых скобок возникнет синтаксическая ошибка или предложение будет восприниматься как относящееся к результату операции над множествами, а не к её аргументам. Например,

```
SELECT a FROM b UNION SELECT x FROM y LIMIT 10
```

не вызовет ошибку, но будет означать

```
(SELECT a FROM b UNION SELECT x FROM y) LIMIT 10
```

а не

```
SELECT a FROM b UNION (SELECT x FROM y LIMIT 10)
```

7.5. Сортировка строк

После того как запрос выдал таблицу результатов (после обработки списка выборки), её можно отсортировать. Если сортировка не задана, строки возвращаются в неопределённом порядке.

Фактический порядок строк в этом случае будет зависеть от плана соединения и сканирования, а также от порядка данных на диске, поэтому полагаться на него нельзя. Определённый порядок выводимых строк гарантируется, только если этап сортировки задан явно.

Порядок сортировки определяет предложение ORDER BY:

```
SELECT список_выборки
    FROM табличное_выражение
    ORDER BY выражение_сортировки1 [ASC | DESC] [NULLS { FIRST | LAST }]
             [, выражение_сортировки2 [ASC | DESC] [NULLS { FIRST | LAST }] ...]
```

Выражениями сортировки могут быть любые выражения, допустимые в списке выборки запроса. Например:

```
SELECT a, b FROM table1 ORDER BY a + b, c;
```

Когда указывается несколько выражений, последующие значения позволяют отсортировать строки, в которых совпали все предыдущие значения. Каждое выражение можно дополнить ключевыми словами ASC или DESC, которые выбирают сортировку соответственно по возрастанию или убыванию. По умолчанию принят порядок по возрастанию (ASC). При сортировке по возрастанию сначала идут меньшие значения, где понятие «меньше» определяется оператором <. Подобным образом, сортировка по возрастанию определяется оператором >. [1]

Для определения места значений NULL можно использовать указания NULLS FIRST и NULLS LAST, которые помещают значения NULL соответственно до или после значений не NULL. По умолчанию значения NULL считаются больше любых других, то есть подразумевается NULLS FIRST для порядка DESC и NULLS LAST в противном случае.

Заметьте, что порядки сортировки определяются независимо для каждого столбца. Например, ORDER BY x, y DESC означает ORDER BY x ASC, y DESC, и это не то же самое, что ORDER BY x DESC, y DESC.

Здесь *выражение_сортировки* может быть меткой столбца или номером выводимого столбца, как в данном примере:

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
```

Оба эти запроса сортируют результат по первому столбцу. Заметьте, что имя выводимого столбца должно оставаться само по себе, его нельзя использовать в выражении. Например, это *ошибка*:

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum + c;          -- неправильно
```

Это ограничение позволяет уменьшить неоднозначность. Тем не менее неоднозначность возможна, когда в ORDER BY указано простое имя, но оно соответствует и имени выходного столбца, и столбцу из табличного выражения. В этом случае используется выходной столбец. Эта ситуация может возникнуть, только когда с помощью AS выходному столбцу назначается то же имя, что имеет столбец в другой таблице.

ORDER BY можно применить к результату комбинации UNION, INTERSECT и EXCEPT, но в этом случае возможна сортировка только по номерам или именам столбцов, но не по выражениям.

7.6. LIMIT и OFFSET

Указания LIMIT и OFFSET позволяют получить только часть строк из тех, что выдал остальной запрос:

```
SELECT список_выборки
    FROM табличное_выражение
```

[1] На деле Postgres Pro определяет порядок сортировки для ASC и DESC по *классу оператора B-дерева по умолчанию* для типа данных выражения. Обычно типы данных создаются так, что этому порядку соответствуют операторы < и >, но возможно разработать собственный тип данных, который будет вести себя по-другому.

```
    [ ORDER BY ... ]
    [ LIMIT { число | ALL } ] [ OFFSET число ]
```

Если указывается число LIMIT, в результате возвращается не больше заданного числа строк (меньше может быть, если сам запрос выдал меньшее количество строк). `LIMIT ALL` равносильно отсутствию указания `LIMIT`, как и `LIMIT` с аргументом NULL.

`OFFSET` указывает пропустить указанное число строк, прежде чем начать выдавать строки. `OFFSET 0` равносильно отсутствию указания `OFFSET`, как и `OFFSET` с аргументом NULL.

Если указано и `OFFSET`, и `LIMIT`, сначала система пропускает `OFFSET` строк, а затем начинает подсчитывать строки для ограничения `LIMIT`.

Применяя `LIMIT`, важно использовать также предложение `ORDER BY`, чтобы строки результата выдавались в определённом порядке. Иначе будут возвращаться непредсказуемые подмножества строк. Вы можете запросить строки с десятой по двадцатую, но какой порядок вы имеете в виду? Порядок будет неизвестен, если не добавить `ORDER BY`.

Оптимизатор запроса учитывает ограничение `LIMIT`, строя планы выполнения запросов, поэтому вероятнее всего планы (а значит и порядок строк) будут меняться при разных `LIMIT` и `OFFSET`. Таким образом, различные значения `LIMIT`/`OFFSET`, выбирающие разные подмножества результатов запроса, *приведут к несогласованности результатов*, если не установить предсказуемую сортировку с помощью `ORDER BY`. Это не ошибка, а неизбежное следствие того, что SQL не гарантирует вывод результатов запроса в некотором порядке, если порядок не определён явно предложением `ORDER BY`.

Строки, пропускаемые согласно предложению `OFFSET`, тем не менее должны вычисляться на сервере. Таким образом, при больших значениях `OFFSET` работает неэффективно.

7.7. Списки VALUES

Предложение `VALUES` позволяет создать «постоянную таблицу», которую можно использовать в запросе, не создавая и не наполняя таблицу в БД. Синтаксис предложения:

```
VALUES ( выражение [, ...] ) [, ...]
```

Для каждого списка выражений в скобках создаётся строка таблицы. Все списки должны иметь одинаковое число элементов (т. е. число столбцов в таблице) и соответствующие элементы во всех списках должны иметь совместимые типы данных. Фактический тип данных столбцов результата определяется по тем же правилам, что и для `UNION` (см. Раздел 10.5).

Как пример:

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

вернёт таблицу из двух столбцов и трёх строк. Это равносильно такому запросу:

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

По умолчанию Postgres Pro назначает столбцам таблицы `VALUES` имена `column1`, `column2` и т. д. Имена столбцов не определены в стандарте SQL и в другой СУБД они могут быть другими, поэтому обычно лучше переопределить имена списком псевдонимов, например так:

```
=> SELECT * FROM (VALUES (1, 'one'), (2, 'two'), (3, 'three')) AS t (num,letter);
 num | letter
-----+--------
   1 | one
   2 | two
```

```
   3 | three
(3 rows)
```

Синтаксически список VALUES с набором выражений равнозначен:

```
SELECT список_выборки FROM табличное_выражение
```

и допускается везде, где допустим SELECT. Например, вы можете использовать его в составе UNION или добавить к нему *определение_сортировки* (ORDER BY, LIMIT и/или OFFSET). VALUES чаще всего используется как источник данных для команды INSERT, а также как подзапрос.

За дополнительными сведениями обратитесь к справке VALUES.

7.8. Запросы WITH (Общие табличные выражения)

WITH предоставляет способ записывать дополнительные операторы для применения в больших запросах. Эти операторы, которые также называют общими табличными выражениями (Common Table Expressions, CTE), можно представить как определения временных таблиц, существующих только для одного запроса. Дополнительным оператором в предложении WITH может быть SELECT, INSERT, UPDATE или DELETE, а само предложение WITH присоединяется к основному оператору, которым также может быть SELECT, INSERT, UPDATE или DELETE.

7.8.1. SELECT в WITH

Основное предназначение SELECT в предложении WITH заключается в разбиении сложных запросов на простые части. Например, запрос:

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
   ), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
   )
SELECT region,
   product,
   SUM(quantity) AS product_units,
   SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

выводит итоги по продажам только для передовых регионов. Предложение WITH определяет два дополнительных оператора regional_sales и top_regions так, что результат regional_sales используется в top_regions, а результат top_regions используется в основном запросе SELECT. Этот пример можно было бы переписать без WITH, но тогда нам понадобятся два уровня вложенных подзапросов SELECT. Показанным выше способом это можно сделать немного проще.

Необязательное указание RECURSIVE превращает WITH из просто удобной синтаксической конструкции в средство реализации того, что невозможно в стандартном SQL. Используя RECURSIVE, запрос WITH может обращаться к собственному результату. Очень простой пример, суммирующий числа от 1 до 100:

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

В общем виде рекурсивный запрос WITH всегда записывается как *не рекурсивная часть*, потом UNION (или UNION ALL), а затем *рекурсивная часть*, где только в рекурсивной части можно обратиться к результату запроса. Такой запрос выполняется следующим образом:

Вычисление рекурсивного запроса

1. Вычисляется не рекурсивная часть. Для UNION (но не UNION ALL) отбрасываются дублирующиеся строки. Все оставшиеся строки включаются в результат рекурсивного запроса и также помещаются во временную *рабочую таблицу*.
2. Пока рабочая таблица не пуста, повторяются следующие действия:
   a. Вычисляется рекурсивная часть так, что рекурсивная ссылка на сам запрос обращается к текущему содержимому рабочей таблицы. Для UNION (но не UNION ALL) отбрасываются дублирующиеся строки и строки, дублирующие ранее полученные. Все оставшиеся строки включаются в результат рекурсивного запроса и также помещаются во временную *промежуточную таблицу*.
   b. Содержимое рабочей таблицы заменяется содержимым промежуточной таблицы, а затем промежуточная таблица очищается.

> **Примечание**
>
> Хотя указание RECURSIVE позволяет определять рекурсивные запросы, внутри такие запросы обрабатываются итерационно.

В показанном выше примере в рабочей таблице на каждом этапе содержится всего одна строка и в ней последовательно накапливаются значения от 1 до 100. На сотом шаге, благодаря условию WHERE, не возвращается ничего, так что вычисление запроса завершается.

Рекурсивные запросы обычно применяются для работы с иерархическими или древовидными структурами данных. В качестве полезного примера можно привести запрос, находящий все непосредственные и косвенные составные части продукта, используя только таблицу с прямыми связями:

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part, p.quantity * pr.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
  )
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

Работая с рекурсивными запросами, важно обеспечить, чтобы рекурсивная часть запроса в конце концов не выдала никаких кортежей (строк), в противном случае цикл будет бесконечным. Иногда для этого достаточно применять UNION вместо UNION ALL, так как при этом будут отбрасываться строки, которые уже есть в результате. Однако часто в цикле выдаются строки, не совпадающие полностью с предыдущими: в таких случаях может иметь смысл проверить одно или несколько полей, чтобы определить, не была ли текущая точка достигнута раньше. Стандартный способ решения подобных задач — вычислить массив с уже обработанными значениями. Например, рассмотрите следующий запрос, просматривающий таблицу graph по полю link:

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (
        SELECT g.id, g.link, g.data, 1
        FROM graph g
      UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1
```

```
        FROM graph g, search_graph sg
        WHERE g.id = sg.link
)
SELECT * FROM search_graph;
```

Этот запрос зациклится, если связи `link` содержат циклы. Так как нам нужно получать в результате «depth», одно лишь изменение `UNION ALL` на `UNION` не позволит избежать зацикливания. Вместо этого мы должны как-то определить, что уже достигали текущей строки, пройдя некоторый путь. Для этого мы добавляем два столбца `path` и `cycle` и получаем запрос, защищённый от зацикливания:

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
        SELECT g.id, g.link, g.data, 1,
          ARRAY[g.id],
          false
        FROM graph g
      UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1,
          path || g.id,
          g.id = ANY(path)
        FROM graph g, search_graph sg
        WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

Помимо предотвращения циклов, значения массива часто бывают полезны сами по себе для представления «пути», приведшего к определённой строке.

В общем случае, когда для выявления цикла нужно проверять несколько полей, следует использовать массив строк. Например, если нужно сравнить поля `f1` и `f2`:

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
        SELECT g.id, g.link, g.data, 1,
          ARRAY[ROW(g.f1, g.f2)],
          false
        FROM graph g
      UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1,
          path || ROW(g.f1, g.f2),
          ROW(g.f1, g.f2) = ANY(path)
        FROM graph g, search_graph sg
        WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

Подсказка

Часто для выявления цикла достаточно одного поля, и тогда `ROW()` можно опустить. При этом будет использоваться не массив данных составного типа, а простой массив, что более эффективно.

Подсказка

Этот алгоритм рекурсивного вычисления запроса выдаёт в результате узлы, упорядоченные «сначала в ширину». Чтобы получить результаты, отсортированные в порядке «сначала в глубину», можно добавить во внешний запрос `ORDER BY` по столбцу «path», полученному, как показано выше.

Для тестирования запросов, которые могут зацикливаться, есть хороший приём — добавить LIMIT в родительский запрос. Например, следующий запрос зациклится, если не добавить предложение LIMIT:

```
WITH RECURSIVE t(n) AS (
    SELECT 1
  UNION ALL
    SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

Но в данном случае этого не происходит, так как в Postgres Pro запрос WITH выдаёт столько строк, сколько фактически принимает родительский запрос. В производственной среде использовать этот приём не рекомендуется, так как другие системы могут вести себя по-другому. Кроме того, это не будет работать, если внешний запрос сортирует результаты рекурсивного запроса или соединяет их с другой таблицей, так как в подобных случаях внешний запрос обычно всё равно выбирает результат запроса WITH полностью.

Запросы WITH имеют полезное свойство — они вычисляются только раз для всего родительского запроса, даже если этот запрос или соседние запросы WITH обращаются к ним неоднократно. Таким образом, сложные вычисления, результаты которых нужны в нескольких местах, можно выносить в запросы WITH в целях оптимизации. Кроме того, такие запросы позволяют избежать нежелательных вычислений функций с побочными эффектами. Однако есть и обратная сторона — оптимизатор не может распространить ограничения родительского запроса на запрос WITH так, как он делает это для обычного подзапроса. Запрос WITH обычно выполняется буквально и возвращает все строки, включая те, что потом может отбросить родительский запрос. (Но как было сказано выше, вычисление может остановиться раньше, если в ссылке на этот запрос затребуется только ограниченное число строк.)

Примеры выше показывают только предложение WITH с SELECT, но таким же образом его можно использовать с командами INSERT, UPDATE и DELETE. В каждом случае он по сути создаёт временную таблицу, к которой можно обратиться в основной команде.

7.8.2. Изменение данных в WITH

В предложении WITH можно также использовать операторы, изменяющие данные (INSERT, UPDATE или DELETE). Это позволяет выполнять в одном запросе сразу несколько разных операций. Например:

```
WITH moved_rows AS (
    DELETE FROM products
    WHERE
        "date" >= '2010-10-01' AND
        "date" < '2010-11-01'
    RETURNING *
)
INSERT INTO products_log
SELECT * FROM moved_rows;
```

Этот запрос фактически перемещает строки из products в products_log. Оператор DELETE в WITH удаляет указанные строки из products и возвращает их содержимое в предложении RETURNING; а затем главный запрос читает это содержимое и вставляет в таблицу products_log.

Следует заметить, что предложение WITH в данном случае присоединяется к оператору INSERT, а не к SELECT, вложенному в INSERT. Это необходимо, так как WITH может содержать операторы, изменяющие данные, только на верхнем уровне запроса. Однако при этом применяются обычные правила видимости WITH, так что к результату WITH можно обратиться и из вложенного оператора SELECT.

Операторы, изменяющие данные, в WITH обычно дополняются предложением RETURNING (см. Раздел 6.4), как показано в этом примере. Важно понимать, что временная таблица, которую

можно будет использовать в остальном запросе, создаётся из результата RETURNING, а *не* целевой таблицы оператора. Если оператор, изменяющий данные, в WITH не дополнен предложением RETURNING, временная таблица не создаётся и обращаться к ней в остальном запросе нельзя. Однако такой запрос всё равно будет выполнен. Например, допустим следующий не очень практичный запрос:

```
WITH t AS (
    DELETE FROM foo
)
DELETE FROM bar;
```

Он удалит все строки из таблиц foo и bar. При этом число задействованных строк, которое получит клиент, будет подсчитываться только по строкам, удалённым из bar.

Рекурсивные ссылки в операторах, изменяющих данные, не допускаются. В некоторых случаях это ограничение можно обойти, обратившись к конечному результату рекурсивного WITH, например так:

```
WITH RECURSIVE included_parts(sub_part, part) AS (
    SELECT sub_part, part FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
  )
DELETE FROM parts
  WHERE part IN (SELECT part FROM included_parts);
```

Этот запрос удаляет все непосредственные и косвенные составные части продукта.

Операторы, изменяющие данные в WITH, выполняются только один раз и всегда полностью, вне зависимости от того, принимает ли их результат основной запрос. Заметьте, что это отличается от поведения SELECT в WITH: как говорилось в предыдущем разделе, SELECT выполняется только до тех пор, пока его результаты востребованы основным запросом.

Вложенные операторы в WITH выполняются одновременно друг с другом и с основным запросом. Таким образом, порядок, в котором операторы в WITH будут фактически изменять данные, непредсказуем. Все эти операторы выполняются с одним *снимком данных* (см. Главу 13), так что они не могут «видеть», как каждый из них меняет целевые таблицы. Это уменьшает эффект непредсказуемости фактического порядка изменения строк и означает, что RETURNING — единственный вариант передачи изменений от вложенных операторов WITH основному запросу. Например, в данном случае:

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM products;
```

внешний оператор SELECT выдаст цены, которые были до действия UPDATE, тогда как в запросе

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM t;
```

внешний SELECT выдаст изменённые данные.

Неоднократное изменение одной и той же строки в рамках одного оператора не поддерживается. Иметь место будет только одно из нескольких изменений и надёжно определить, какое именно, часто довольно сложно (а иногда и вовсе невозможно). Это так же касается случая, когда

строка удаляется и изменяется в том же операторе: в результате может быть выполнено только обновление. Поэтому в общем случае следует избегать подобного наложения операций. В частности, избегайте подзапросов `WITH`, которые могут повлиять на строки, изменяемые основным оператором или операторами, вложенные в него. Результат действия таких запросов будет непредсказуемым.

В настоящее время, для оператора, изменяющего данные в `WITH`, в качестве целевой нельзя использовать таблицу, для которой определено условное правило или правило `ALSO` или `INSTEAD`, если оно состоит из нескольких операторов.

Глава 8. Типы данных

Postgres Pro предоставляет пользователям богатый ассортимент встроенных типов данных. Кроме того, пользователи могут создавать свои типы в Postgres Pro, используя команду CREATE TYPE.

Таблица 8.1 содержит все встроенные типы данных общего пользования. Многие из альтернативных имён, приведённых в столбце «Псевдонимы», используются внутри Postgres Pro по историческим причинам. В этот список не включены некоторые устаревшие типы и типы для внутреннего применения.

Таблица 8.1. Типы данных

Имя	Псевдонимы	Описание
bigint	int8	знаковое целое из 8 байт
bigserial	serial8	восьмибайтное целое с автоувеличением
bit [(*n*)]		битовая строка фиксированной длины
bit varying [(*n*)]	varbit [(*n*)]	битовая строка переменной длины
boolean	bool	логическое значение (true/false)
box		прямоугольник в плоскости
bytea		двоичные данные («массив байт»)
character [(*n*)]	char [(*n*)]	символьная строка фиксированной длины
character varying [(*n*)]	varchar [(*n*)]	символьная строка переменной длины
cidr		сетевой адрес IPv4 или IPv6
circle		круг в плоскости
date		календарная дата (год, месяц, день)
double precision	float8	число двойной точности с плавающей точкой (8 байт)
inet		адрес узла IPv4 или IPv6
integer	int, int4	знаковое четырёхбайтное целое
interval [*поля*] [(*p*)]		интервал времени
json		текстовые данные JSON
jsonb		двоичные данные JSON, разобранные
line		прямая в плоскости
lseg		отрезок в плоскости
macaddr		MAC-адрес
macaddr8		адрес MAC (Media Access Control) (в формате EUI-64)
money		денежная сумма
numeric [(*p*, *s*)]	decimal [(*p*, *s*)]	вещественное число заданной точности

Имя	Псевдонимы	Описание
path		геометрический путь в плоскости
pg_lsn		последовательный номер в журнале Postgres Pro
point		геометрическая точка в плоскости
polygon		замкнутый геометрический путь в плоскости
real	float4	число одинарной точности с плавающей точкой (4 байта)
smallint	int2	знаковое двухбайтное целое
smallserial	serial2	двухбайтное целое с автоувеличением
serial	serial4	четырёхбайтное целое с автоувеличением
text		символьная строка переменной длины
time [(*p*)] [without time zone]		время суток (без часового пояса)
time [(*p*)] with time zone	timetz	время суток с учётом часового пояса
timestamp [(*p*)] [without time zone]		дата и время (без часового пояса)
timestamp [(*p*)] with time zone	timestamptz	дата и время с учётом часового пояса
tsquery		запрос текстового поиска
tsvector		документ для текстового поиска
txid_snapshot		снимок идентификатора транзакций
uuid		универсальный уникальный идентификатор
xml		XML-данные

Совместимость

В стандарте SQL описаны следующие типы (или их имена): bigint, bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time (с часовым поясом и без), timestamp (с часовым поясом и без), xml.

Каждый тип данных имеет внутреннее представление, скрытое функциями ввода и вывода. При этом многие встроенные типы стандартны и имеют очевидные внешние форматы. Однако есть типы, уникальные для Postgres Pro, например геометрические пути, и есть типы, которые могут иметь разные форматы, например, дата и время. Некоторые функции ввода и вывода не являются в точности обратными друг к другу, то есть результат функции вывода может не совпадать со входным значением из-за потери точности.

8.1. Числовые типы

Числовые типы включают двух-, четырёх- и восьмибайтные целые, четырёх- и восьмибайтные числа с плавающей точкой, а также десятичные числа с задаваемой точностью. Все эти типы перечислены в Таблице 8.2.

Таблица 8.2. Числовые типы

Имя	Размер	Описание	Диапазон
`smallint`	2 байта	целое в небольшом диапазоне	-32768 .. +32767
`integer`	4 байта	типичный выбор для целых чисел	-2147483648 .. +2147483647
`bigint`	8 байт	целое в большом диапазоне	-9223372036854775808. 9223372036854775807
`decimal`	переменный	вещественное число с указанной точностью	до 131072 цифр до десятичной точки и до 16383 — после
`numeric`	переменный	вещественное число с указанной точностью	до 131072 цифр до десятичной точки и до 16383 — после
`real`	4 байта	вещественное число с переменной точностью	точность в пределах 6 десятичных цифр
`double precision`	8 байт	вещественное число с переменной точностью	точность в пределах 15 десятичных цифр
`smallserial`	2 байта	небольшое целое с автоувеличением	1 .. 32767
`serial`	4 байта	целое с автоувеличением	1 .. 2147483647
`bigserial`	8 байт	большое целое с автоувеличением	1 .. 9223372036854775807

Синтаксис констант числовых типов описан в Подразделе 4.1.2. Для этих типов определён полный набор соответствующих арифметических операторов и функций. За дополнительными сведениями обратитесь к Главе 9. Подробнее эти типы описаны в следующих разделах.

8.1.1. Целочисленные типы

Типы `smallint`, `integer` и `bigint` хранят целые числа, то есть числа без дробной части, имеющие разные допустимые диапазоны. Попытка сохранить значение, выходящее за рамки диапазона, приведёт к ошибке.

Чаще всего используется тип `integer`, как наиболее сбалансированный выбор ширины диапазона, размера и быстродействия. Тип `smallint` обычно применяется, только когда крайне важно уменьшить размер данных на диске. Тип `bigint` предназначен для тех случаев, когда числа не умещаются в диапазон типа `integer`.

В SQL определены только типы `integer` (или `int`), `smallint` и `bigint`. Имена типов `int2`, `int4` и `int8` выходят за рамки стандарта, хотя могут работать и в некоторых других СУБД.

8.1.2. Числа с произвольной точностью

Тип `numeric` позволяет хранить числа с очень большим количеством цифр. Он особенно рекомендуется для хранения денежных сумм и других величин, где важна точность. Вычисления с типом `numeric` дают точные результаты, где это возможно, например, при сложении, вычитании

и умножении. Однако операции со значениями numeric выполняются гораздо медленнее, чем с целыми числами или с типами с плавающей точкой, описанными в следующем разделе.

Ниже мы используем следующие термины: *масштаб* значения numeric определяет количество десятичных цифр в дробной части, справа от десятичной точки, а *точность* — общее количество значимых цифр в числе, т. е. количество цифр по обе стороны десятичной точки. Например, число 23.5141 имеет точность 6 и масштаб 4. Целочисленные значения можно считать числами с масштабом 0.

Для столбца типа numeric можно настроить и максимальную точность, и максимальный масштаб. Столбец типа numeric объявляется следующим образом:

NUMERIC(*точность*, *масштаб*)

Точность должна быть положительной, а масштаб положительным или равным нулю. Альтернативный вариант

NUMERIC(*точность*)

устанавливает масштаб 0. Форма:

NUMERIC

без указания точности и масштаба создаёт столбец, в котором можно сохранять числовые значения любой точности и масштаба в пределах, поддерживаемых системой. В столбце этого типа входные значения не будут приводиться к какому-либо масштабу, тогда как в столбцах numeric с явно заданным масштабом значения подгоняются под этот масштаб. (Стандарт SQL утверждает, что по умолчанию должен устанавливаться масштаб 0, т. е. значения должны приводиться к целым числам. Однако мы считаем это не очень полезным. Если для вас важна переносимость, всегда указывайте точность и масштаб явно.)

> **Примечание**
>
> Максимально допустимая точность, которую можно указать в объявлении типа, равна 1000; если же использовать NUMERIC без указания точности, действуют ограничения, описанные в Таблице 8.2.

Если масштаб значения, которое нужно сохранить, превышает объявленный масштаб столбца, система округлит его до заданного количества цифр после точки. Если же после этого количество цифр слева в сумме с масштабом превысит объявленную точность, произойдёт ошибка.

Числовые значения физически хранятся без каких-либо дополняющих нулей слева или справа. Таким образом, объявляемые точность и масштаб столбца определяют максимальный, а не фиксированный размер хранения. (В этом смысле тип numeric больше похож на тип varchar(*n*), чем на char(*n*).) Действительный размер хранения такого значения складывается из двух байт для каждой группы из четырёх цифр и дополнительных трёх-восьми байт.

Помимо обычных чисел тип numeric позволяет сохранить специальное значение NaN, что означает «not-a-number» (не число). Любая операция с NaN выдаёт в результате тоже NaN. Записывая это значение в виде константы в команде SQL, его нужно заключать в апострофы, например так: UPDATE table SET x = 'NaN'. Регистр символов в строке NaN не важен.

> **Примечание**
>
> В большинстве реализаций «не число» (NaN) считается не равным любому другому значению (в том числе и самому NaN). Чтобы значения numeric можно было сортировать и использовать в древовидных индексах, Postgres Pro считает, что значения NaN равны друг другу и при этом больше любых числовых значений (не NaN).

Типы `decimal` и `numeric` равнозначны. Оба эти типа описаны в стандарте SQL.

При округлении значений тип `numeric` выдаёт число, большее по модулю, тогда как (на большинстве платформ) типы `real` и `double precision` выдают ближайшее чётное число. Например:

```
SELECT x,
  round(x::numeric) AS num_round,
  round(x::double precision) AS dbl_round
FROM generate_series(-3.5, 3.5, 1) as x;
  x   | num_round | dbl_round
------+-----------+-----------
 -3.5 |        -4 |        -4
 -2.5 |        -3 |        -2
 -1.5 |        -2 |        -2
 -0.5 |        -1 |        -0
  0.5 |         1 |         0
  1.5 |         2 |         2
  2.5 |         3 |         2
  3.5 |         4 |         4
(8 rows)
```

8.1.3. Типы с плавающей точкой

Типы данных `real` и `double precision` хранят приближённые числовые значения с переменной точностью. На практике эти типы обычно реализуют стандарт IEEE 754 для двоичной арифметики с плавающей точкой (с одинарной и двойной точностью соответственно), в той мере, в какой его поддерживают процессор, операционная система и компилятор.

Неточность здесь выражается в том, что некоторые значения, которые нельзя преобразовать во внутренний формат, сохраняются приближённо, так что полученное значение может несколько отличаться от записанного. Управление подобными ошибками и их распространение в процессе вычислений является предметом изучения целого раздела математики и компьютерной науки, и здесь не рассматривается. Мы отметим только следующее:

- Если вам нужна точность при хранении и вычислениях (например, для денежных сумм), используйте вместо этого тип `numeric`.
- Если вы хотите выполнять с этими типами сложные вычисления, имеющие большую важность, тщательно изучите реализацию операций в вашей среде и особенно поведение в крайних случаях (бесконечность, антипереполнение).
- Проверка равенства двух чисел с плавающей точкой может не всегда давать ожидаемый результат.

На большинстве платформ тип `real` может сохранить значения в пределах от 1E-37 до 1E+37 с точностью не меньше 6 десятичных цифр. Тип `double precision` предлагает значения в диапазоне от 1E-307 до 1E+308 и с точностью не меньше 15 цифр. Попытка сохранить слишком большие или слишком маленькие значения приведёт к ошибке. Если точность вводимого числа слишком велика, оно будет округлено. При попытке сохранить число, близкое к 0, но непредставимое как отличное от 0, произойдёт ошибка антипереполнения.

Примечание

Параметр extra_float_digits определяет количество дополнительных значащих цифр при преобразовании значения с плавающей точкой в текст для вывода. Со значением по умолчанию (`0`) вывод будет одинаковым на всех платформах, поддерживаемых Postgres Pro. При его увеличении выводимое значение числа будет более точно представлять хранимое, но от этого может пострадать переносимость.

В дополнение к обычным числовым значениям типы с плавающей точкой принимают следующие специальные значения:

```
Infinity
-Infinity
NaN
```

Они представляют особые значения, описанные в IEEE 754, соответственно «бесконечность», «минус бесконечность» и «не число». (На компьютерах, где арифметика с плавающей точкой не соответствует стандарту IEEE 754, эти значения, вероятно, не будут работать должным образом.) Записывая эти значения в виде констант в команде SQL, их нужно заключать в апострофы, например так: `UPDATE table SET x = '-Infinity'`. Регистр символов в этих строках не важен.

> **Примечание**
>
> Согласно IEEE754, `NaN` не должно считаться равным любому другому значению с плавающей точкой (в том числе и самому `NaN`). Чтобы значения с плавающей точкой можно было сортировать и использовать в древовидных индексах, Postgres Pro считает, что значения `NaN` равны друг другу, и при этом больше любых числовых значений (не `NaN`).

Postgres Pro также поддерживает форматы `float` и `float(`*p*`)`, оговорённые в стандарте SQL, для указания неточных числовых типов. Здесь *p* определяет минимально допустимую точность в *двоичных* цифрах. Postgres Pro воспринимает запись от `float(1)` до `float(24)` как выбор типа `real`, а запись от `float(25)` до `float(53)` как выбор типа `double precision`. Значения *p* вне допустимого диапазона вызывают ошибку. Если `float` указывается без точности, подразумевается тип `double precision`.

> **Примечание**
>
> Предположение, что типы `real` и `double precision` имеют в мантиссе 24 и 53 бита соответственно, справедливо для всех реализаций плавающей точки по стандарту IEEE. На платформах, не поддерживающих IEEE, размер мантиссы может несколько отличаться, но для простоты диапазоны *p* везде считаются одинаковыми.

8.1.4. Последовательные типы

> **Примечание**
>
> В этом разделе описывается специфичный для Postgres Pro способ создания столбца с автоувеличением. Другой способ, соответствующий стандарту SQL, заключается в использовании столбцов идентификации и рассматривается в описании CREATE TABLE.

Типы данных `smallserial`, `serial` и `bigserial` не являются настоящими типами, а представляют собой просто удобное средство для создания столбцов с уникальными идентификаторами (подобное свойству `AUTO_INCREMENT` в некоторых СУБД). В текущей реализации запись:

```
CREATE TABLE имя_таблицы (
    имя_столбца SERIAL
);
```

равнозначна следующим командам:

```
CREATE SEQUENCE имя_таблицы_имя_столбца_seq AS integer;
CREATE TABLE имя_таблицы (
    имя_столбца integer NOT NULL DEFAULT nextval('имя_таблицы_имя_столбца_seq')
);
```

```
ALTER SEQUENCE имя_таблицы_имя_столбца_seq OWNED BY имя_таблицы.имя_столбца;
```

То есть при определении такого типа создаётся целочисленный столбец со значением по умолчанию, извлекаемым из генератора последовательности. Чтобы в столбец нельзя было вставить NULL, в его определение добавляется ограничение `NOT NULL`. (Во многих случаях также имеет смысл добавить для этого столбца ограничения `UNIQUE` или `PRIMARY KEY` для защиты от ошибочного добавления дублирующихся значений, но автоматически это не происходит.) Последняя команда определяет, что последовательность «принадлежит» столбцу, так что она будет удалена при удалении столбца или таблицы.

> **Примечание**
>
> Так как типы `smallserial`, `serial` и `bigserial` реализованы через последовательности, в числовом ряду значений столбца могут образовываться пропуски (или «дыры»), даже если никакие строки не удалялись. Значение, выделенное из последовательности, считается «задействованным», даже если строку с этим значением не удалось вставить в таблицу. Это может произойти, например, при откате транзакции, добавляющей данные. См. описание `nextval()` в Разделе 9.16.

Чтобы вставить в столбец `serial` следующее значение последовательности, ему нужно присвоить значение по умолчанию. Это можно сделать, либо исключив его из списка столбцов в операторе `INSERT`, либо с помощью ключевого слова `DEFAULT`.

Имена типов `serial` и `serial4` равнозначны: они создают столбцы `integer`. Так же являются синонимами имена `bigserial` и `serial8`, но они создают столбцы `bigint`. Тип `bigserial` следует использовать, если за всё время жизни таблицы планируется использовать больше чем 2^{31} значений. И наконец, синонимами являются имена типов `smallserial` и `serial2`, но они создают столбец `smallint`.

Последовательность, созданная для столбца `serial`, автоматически удаляется при удалении связанного столбца. Последовательность можно удалить и отдельно от столбца, но при этом также будет удалено определение значения по умолчанию.

8.2. Денежные типы

Тип `money` хранит денежную сумму с фиксированной дробной частью; см. Таблицу 8.3. Точность дробной части определяется на уровне базы данных параметром lc_monetary. Для диапазона, показанного в таблице, предполагается, что число содержит два знака после запятой. Входные данные могут быть записаны по-разному, в том числе в виде целых и дробных чисел, а также в виде строки в денежном формате, например `'$1,000.00'`. Выводятся эти значения обычно в денежном формате, зависящем от региональных стандартов.

Таблица 8.3. Денежные типы

Имя	Размер	Описание	Диапазон
`money`	8 байт	денежная сумма	-92233720368547 758.08 .. +92233720368547 758.07

Так как выводимые значения этого типа зависят от региональных стандартов, попытка загрузить данные типа `money` в базу данных с другим параметром `lc_monetary` может быть неудачной. Во избежание подобных проблем, прежде чем восстанавливать копию в новую базу данных, убедитесь в том, что параметр `lc_monetary` в этой базе данных имеет то же значение, что и в исходной.

Значения типов `numeric`, `int` и `bigint` можно привести к типу `money`. Преобразования типов `real` и `double precision` так же возможны через тип `numeric`, например:

```
SELECT '12.34'::float8::numeric::money;
```

Однако использовать числа с плавающей точкой для денежных сумм не рекомендуется из-за возможных ошибок округления.

Значение `money` можно привести к типу `numeric` без потери точности. Преобразование в другие типы может быть неточным и также должно выполняться в два этапа:

```
SELECT '52093.89'::money::numeric::float8;
```

При делении значения типа `money` на целое число выполняется отбрасывание дробной части и получается целое, ближайшее к нулю. Чтобы получить результат с округлением, выполните деление значения с плавающей точкой или приведите значение типа `money` к `numeric` до деления, а затем приведите результат к типу `money`. (Последний вариант предпочтительнее, так как исключает риск потери точности.) Когда значение `money` делится на другое значение `money`, результатом будет значение типа `double precision` (то есть просто число, не денежная величина); денежные единицы измерения при делении сокращаются.

8.3. Символьные типы

Таблица 8.4. Символьные типы

Имя	Описание
`character varying( n), varchar( n)`	строка ограниченной переменной длины
`character( n), char( n)`	строка фиксированной длины, дополненная пробелами
`text`	строка неограниченной переменной длины

В Таблице 8.4 перечислены символьные типы общего назначения, доступные в Postgres Pro.

SQL определяет два основных символьных типа: `character varying(`*n*`)` и `character(`*n*`)`, где *n* — положительное число. Оба эти типа могут хранить текстовые строки длиной до *n* символов (не байт). Попытка сохранить в столбце такого типа более длинную строку приведёт к ошибке, если только все лишние символы не являются пробелами (тогда они будут усечены до максимально допустимой длины). (Это несколько странное исключение продиктовано стандартом SQL.) Если длина сохраняемой строки оказывается меньше объявленной, значения типа `character` будут дополнятся пробелами; а тип `character varying` просто сохранит короткую строку.

При попытке явно привести значение к типу `character varying(`*n*`)` или `character(`*n*`)`, часть строки, выходящая за границу в *n* символов, удаляется, не вызывая ошибки. (Это также продиктовано стандартом SQL.)

Записи `varchar(`*n*`)` и `char(`*n*`)` являются синонимами `character varying(`*n*`)` и `character(`*n*`)` соответственно. Если *n* задано, оно должно быть больше нуля и меньше или равно 10485760. Записи `character` без указания длины соответствует `character(1)`. Если же длина не указывается для `character varying`, этот тип будет принимать строки любого размера. Это поведение является расширением Postgres Pro.

Помимо этого, Postgres Pro предлагает тип `text`, в котором можно хранить строки произвольной длины. Хотя тип `text` не описан в стандарте SQL, его поддерживают и некоторые другие СУБД SQL.

Значения типа `character` физически дополняются пробелами до *n* символов и хранятся, а затем отображаются в таком виде. Однако при сравнении двух значений типа `character` дополняющие пробелы считаются незначащими и игнорируются. С правилами сортировки, где пробельные символы являются значащими, это поведение может приводить к неожиданным результатам, например `SELECT 'a '::CHAR(2) collate "C" < E'a\n'::CHAR(2)` вернёт true (условие будет истинным), хотя в локали C символ пробела считается больше символа новой

строки. При приведении значения `character` к другому символьному типу дополняющие пробелы отбрасываются. Заметьте, что эти пробелы *несут* смысловую нагрузку в типах `character varying` и `text` и в проверках по шаблонам, то есть в `LIKE` и регулярных выражениях.

Какие именно символы можно сохранить в этих типах данных, зависит от того, какой набор символов был выбран при создании базы данных. Однако символ с кодом 0 (иногда называемый NUL) сохранить нельзя, вне зависимости от выбранного набора символов. За подробностями обратитесь к Разделу 22.3.

Для хранения короткой строки (до 126 байт) требуется дополнительный 1 байт плюс размер самой строки, включая дополняющие пробелы для типа `character`. Для строк длиннее требуется не 1, а 4 дополнительных байта. Система может автоматически сжимать длинные строки, так что физический размер на диске может быть меньше. Очень длинные текстовые строки переносятся в отдельные таблицы, чтобы они не замедляли работу с другими столбцами. В любом случае максимально возможный размер строки составляет около 1 ГБ. (Допустимое значение *n* в объявлении типа данных меньше этого числа. Это объясняется тем, что в зависимости от кодировки каждый символ может занимать несколько байт. Если вы желаете сохранять строки без определённого предела длины, используйте типы `text` или `character varying` без указания длины, а не задавайте какое-либо большое максимальное значение.)

> **Подсказка**
>
> По быстродействию эти три типа практически не отличаются друг от друга, не считая большего размера хранения для типа с дополняющими пробелами и нескольких машинных операций для проверки длины при сохранении строк в столбце с ограниченной длиной. Хотя в некоторых СУБД тип `character(n)` работает быстрее других, в Postgres Pro это не так; на деле `character(n)` обычно оказывается медленнее остальных типов из-за большего размера данных и более медленной сортировки. В большинстве случаев вместо него лучше применять `text` или `character varying`.

За информацией о синтаксисе строковых констант обратитесь к Подразделу 4.1.2.1, а об имеющихся операторах и функциях вы можете узнать в Главе 9.

Пример 8.1. Использование символьных типов

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- 1

  a   | char_length
------+-------------
 ok   |           2


CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good      ');
INSERT INTO test2 VALUES ('too long');
ОШИБКА:  значение не умещается в тип character varying(5)
INSERT INTO test2 VALUES ('too long'::varchar(5)); -- явное усечение
SELECT b, char_length(b) FROM test2;

   b    | char_length
--------+-------------
 ok     |           2
 good   |           5
 too l  |           5
```

1 Функция char_length рассматривается в Разделе 9.4.

В Postgres Pro есть ещё два символьных типа фиксированной длины, приведённые в Таблице 8.5. Тип name создан *только* для хранения идентификаторов во внутренних системных таблицах и не предназначен для обычного применения пользователями. В настоящее время его длина составляет 64 байта (63 ASCII-символа плюс конечный знак), но в исходном коде C она задаётся константой NAMEDATALEN. Эта константа определяется во время компиляции (и её можно менять в особых случаях), а кроме того, максимальная длина по умолчанию может быть увеличена в следующих версиях. Тип "char" (обратите внимание на кавычки) отличается от char(1) тем, что он фактически хранится в одном байте. Он используется во внутренних системных таблицах для простых перечислений.

Таблица 8.5. Специальные символьные типы

Имя	Размер	Описание
"char"	1 байт	внутренний однобайтный тип
name	64 байта	внутренний тип для имён объектов

8.4. Двоичные типы данных

Для хранения двоичных данных предназначен тип bytea; см. Таблицу 8.6.

Таблица 8.6. Двоичные типы данных

Имя	Размер	Описание
bytea	1 или 4 байта плюс сама двоичная строка	двоичная строка переменной длины

Двоичные строки представляют собой последовательность октетов (байт) и имеют два отличия от текстовых строк. Во-первых, в двоичных строках можно хранить байты с кодом 0 и другими «непечатаемыми» значениями (обычно это значения вне десятичного диапазона 32..126). В текстовых строках нельзя сохранять нулевые байты, а также значения и последовательности значений, не соответствующие выбранной кодировке базы данных. Во-вторых, в операциях с двоичными строками обрабатываются байты в чистом виде, тогда как текстовые строки обрабатываются в зависимости от языковых стандартов. То есть, двоичные строки больше подходят для данных, которые программист видит как «просто байты», а символьные строки — для хранения текста.

Тип bytea поддерживает два формата ввода и вывода: «шестнадцатеричный» и традиционный для PostgreSQL формат «спецпоследовательностей». Входные данные принимаются в обоих форматах, а формат выходных данных зависит от параметра конфигурации bytea_output; по умолчанию выбран шестнадцатеричный. (Заметьте, что шестнадцатеричный формат был введён в PostgreSQL 9.0; в ранних версиях и некоторых программах он не будет работать.)

Стандарт SQL определяет другой тип двоичных данных, BLOB (BINARY LARGE OBJECT, большой двоичный объект). Его входной формат отличается от форматов bytea, но функции и операторы в основном те же.

8.4.1. Шестнадцатеричный формат bytea

В «шестнадцатеричном» формате двоичные данные кодируются двумя шестнадцатеричными цифрами на байт, при этом первая цифра соответствует старшим 4 битам. К полученной строке добавляется префикс \x (чтобы она отличалась от формата спецпоследовательности). В некоторых контекстах обратную косую черту нужно экранировать, продублировав её (см. Подраздел 4.1.2.1). Вводимые шестнадцатеричные цифры могут быть в любом регистре, а между парами цифр допускаются пробельные символы (но не внутри пары и не в начале последовательности \x).

Этот формат совместим со множеством внешних приложений и протоколов, к тому же обычно преобразуется быстрее, поэтому предпочтительнее использовать его.

Пример:

```
SELECT '\xDEADBEEF';
```

8.4.2. Формат спецпоследовательностей `bytea`

Формат «спецпоследовательностей» традиционно использовался в Postgres Pro для значений типа `bytea`. В нём двоичная строка представляется в виде последовательности ASCII-символов, а байты, непредставимые в виде ASCII-символов, передаются в виде спецпоследовательностей. Этот формат может быть удобен, если с точки зрения приложения представление байт в виде символов имеет смысл. Но на практике это обычно создаёт путаницу, так как двоичные и символьные строки могут выглядеть одинаково, а кроме того выбранный механизм спецпоследовательностей довольно неуклюж. Поэтому в новых приложениях этот формат обычно не стоит использовать.

Передавая значения `bytea` в формате спецпоследовательности, байты с определёнными значениями *необходимо* записывать специальным образом, хотя так *можно* записывать и все значения. В общем виде для этого значение байта нужно преобразовать в трёхзначное восьмеричное число и добавить перед ним обратную косую черту. Саму обратную косую черту (символ с десятичным кодом 92) можно записать в виде двух таких символов. В Таблице 8.7 перечислены символы, которые нужно записывать спецпоследовательностями, и приведены альтернативные варианты записи, если они возможны.

Таблица 8.7. Спецпоследовательности записи значений `bytea`

Десятичное значение байта	Описание	Спецпоследовательность ввода	Пример	Шестнадцатеричное представление
0	нулевой байт	`'\000'`	`SELECT '\000'::bytea;`	`\x00`
39	апостроф	`''''` или `'\047'`	`SELECT ''''::bytea;`	`\x27`
92	обратная косая черта	`'\\'` или `'\134'`	`SELECT '\\'::bytea;`	`\x5c`
от 0 до 31 и от 127 до 255	«непечатаемые» байты	`E'\\xxx'` (*xxx* — восьмеричное значение)	`SELECT '\001'::bytea;`	`\x01`

Требования экранирования *непечатаемых* символов определяются языковыми стандартами. Иногда такие символы могут восприниматься и без спецпоследовательностей.

Апострофы должны дублироваться, как показано в Таблице 8.7, потому что это обязательно для любой текстовой строки в команде SQL. При общем разборе текстовой строки внешние апострофы убираются, а каждая пара внутренних сводится к одному символу. Таким образом, функция ввода `bytea` видит всего один апостроф, который она обрабатывает как обычный символ в данных. Дублировать же обратную косую черту при вводе `bytea` не требуется: этот символ считается особым и меняет поведение функции ввода, как показано в Таблице 8.7.

В некоторых контекстах обратная косая черта должна дублироваться (относительно примеров выше), так как при общем разборе строковых констант пара таких символов будет сведена к одному; см. Подраздел 4.1.2.1.

Данные `Bytea` по умолчанию выводятся в шестнадцатеричном формате (`hex`). Если поменять значение bytea_output на `escape`, «непечатаемые» байты представляются в виде соответствующих трёхзначных восьмеричных значений, которые предваряются одной обратной косой чертой. Большинство «печатаемых» байтов представляются обычными символами из клиентского набора символов, например:

```
SET bytea_output = 'escape';

SELECT 'abc \153\154\155 \052\251\124'::bytea;
     bytea
----------------
 abc klm *\251T
```

Байт с десятичным кодом 92 (обратная косая черта) при выводе дублируется. Это иллюстрирует Таблица 8.8.

Таблица 8.8. Спецпоследовательности выходных значений `bytea`

Десятичное значение байта	Описание	Спецпоследовательность вывода	Пример	Выводимый результат
92	обратная косая черта	`\\`	`SELECT '\134'::bytea;`	`\\`
от 0 до 31 и от 127 до 255	«непечатаемые» байты	`\`*xxx* (значение байта)	`SELECT '\001'::bytea;`	`\001`
от 32 до 126	«печатаемые» байты	представление из клиентского набора символов	`SELECT '\176'::bytea;`	~

В зависимости от применяемой клиентской библиотеки Postgres Pro, для преобразования значений `bytea` в спецстроки и обратно могут потребоваться дополнительные действия. Например, если приложение сохраняет в строках символы перевода строк, возможно их также нужно будет представить спецпоследовательностями.

8.5. Типы даты/времени

Postgres Pro поддерживает полный набор типов даты и времени SQL, показанный в Таблице 8.9. Операции, возможные с этими типами данных, описаны в Разделе 9.9. Все даты считаются по Григорианскому календарю, даже для времени до его введения (за дополнительными сведениями обратитесь к Разделу B.6).

Таблица 8.9. Типы даты/времени

Имя	Размер	Описание	Наименьшее значение	Наибольшее значение	Точность
`timestamp [ (`*p*`) ] [ without time zone ]`	8 байт	дата и время (без часового пояса)	4713 до н. э.	294276 н. э.	1 микросекунда
`timestamp [ (`*p*`) ] with time zone`	8 байт	дата и время (с часовым поясом)	4713 до н. э.	294276 н. э.	1 микросекунда
`date`	4 байта	дата (без времени суток)	4713 до н. э.	5874897 н. э.	1 день
`time [ (`*p*`) ] [ without time zone ]`	8 байт	время суток (без даты)	00:00:00	24:00:00	1 микросекунда
`time [ (`*p*`) ] with time zone`	12 байт	время дня (без даты), с часовым поясом	00:00:00+1559	24:00:00-1559	1 микросекунда
`interval [`*поля*`] [ (`*p*`) ]`	16 байт	временной интервал	-178000000 лет	178000000 лет	1 микросекунда

> **Примечание**
>
> Стандарт SQL требует, чтобы тип `timestamp` подразумевал `timestamp without time zone` (время без часового пояса), и Postgres Pro следует этому. Для краткости `timestamp with time zone` можно записать как `timestamptz`; это расширение Postgres Pro.

Типы `time`, `timestamp` и `interval` принимают необязательное значение точности *p*, определяющее, сколько знаков после запятой должно сохраняться в секундах. По умолчанию точность не ограничивается. Допустимые значения *p* лежат в интервале от 0 до 6.

Тип `interval` дополнительно позволяет ограничить набор сохраняемых полей следующими фразами:

```
YEAR
MONTH
DAY
HOUR
MINUTE
SECOND
YEAR TO MONTH
DAY TO HOUR
DAY TO MINUTE
DAY TO SECOND
HOUR TO MINUTE
HOUR TO SECOND
MINUTE TO SECOND
```

Заметьте, что если указаны и *поля*, и точность *p*, указание *поля* должно включать `SECOND`, так как точность применима только к секундам.

Тип `time with time zone` определён стандартом SQL, но в его определении описаны свойства сомнительной ценности. В большинстве случаев сочетание типов `date`, `time`, `timestamp without time zone` и `timestamp with time zone` удовлетворяет все потребности в функционале дат/времени, возникающие в приложениях.

Типы `abstime` и `reltime` имеют меньшую точность и предназначены для внутреннего использования. Эти типы не рекомендуется использовать в обычных приложениях; их может не быть в будущих версиях.

8.5.1. Ввод даты/времени

Значения даты и времени принимаются практически в любом разумном формате, включая ISO 8601, SQL-совместимый, традиционный формат POSTGRES и другие. В некоторых форматах порядок даты, месяца и года во вводимой дате неоднозначен и поэтому поддерживается явное определение формата. Для этого предназначен параметр DateStyle. Когда он имеет значение `MDY`, выбирается интерпретация месяц-день-год, значению `DMY` соответствует день-месяц-год, а `YMD` — год-месяц-день.

Postgres Pro обрабатывает вводимые значения даты/времени более гибко, чем того требует стандарт SQL. Точные правила разбора даты/времени и распознаваемые текстовые поля, в том числе названия месяцев, дней недели и часовых поясов описаны в Приложении B.

Помните, что любые вводимые значения даты и времени нужно заключать в апострофы, как текстовые строки. За дополнительной информацией обратитесь к Подразделу 4.1.2.7. SQL предусматривает следующий синтаксис:

тип [(*p*)] '*значение*'

Здесь *p* — необязательное указание точности, определяющее число знаков после точки в секундах. Точность может быть определена для типов `time`, `timestamp` и `interval` в интервале от 0 до 6. Если

в определении константы точность не указана, она считается равной точности значения в строке (но не больше 6 цифр).

8.5.1.1. Даты

В Таблице 8.10 приведены некоторые допустимые значения типа `date`.

Таблица 8.10. Вводимые даты

Пример	Описание
1999-01-08	ISO 8601; 8 января в любом режиме (рекомендуемый формат)
January 8, 1999	воспринимается однозначно в любом режиме `datestyle`
1/8/1999	8 января в режиме `MDY` и 1 августа в режиме `DMY`
1/18/1999	18 января в режиме `MDY`; недопустимая дата в других режимах
01/02/03	2 января 2003 г. в режиме `MDY`; 1 февраля 2003 г. в режиме `DMY` и 3 февраля 2001 г. в режиме `YMD`
1999-Jan-08	8 января в любом режиме
Jan-08-1999	8 января в любом режиме
08-Jan-1999	8 января в любом режиме
99-Jan-08	8 января в режиме `YMD`; ошибка в других режимах
08-Jan-99	8 января; ошибка в режиме `YMD`
Jan-08-99	8 января; ошибка в режиме `YMD`
19990108	ISO 8601; 8 января 1999 в любом режиме
990108	ISO 8601; 8 января 1999 в любом режиме
1999.008	год и день года
J2451187	юлианский день
January 8, 99 BC	99 до н. э.

8.5.1.2. Время

Для хранения времени суток без даты предназначены типы `time [ (`*p*`) ] without time zone` и `time [ (`*p*`) ] with time zone`. Тип `time` без уточнения эквивалентен типу `time without time zone`.

Допустимые вводимые значения этих типов состоят из записи времени суток и необязательного указания часового пояса. (См. Таблицу 8.11 и Таблицу 8.12.) Если в значении для типа `time without time zone` указывается часовой пояс, он просто игнорируется. Так же будет игнорироваться дата, если её указать, за исключением случаев, когда в указанном часовом поясе принят переход на летнее время, например `America/New_York`. В данном случае указать дату необходимо, чтобы система могла определить, применяется ли обычное или летнее время. Соответствующее смещение часового пояса записывается в значении `time with time zone`.

Таблица 8.11. Вводимое время

Пример	Описание
`04:05:06.789`	ISO 8601
`04:05:06`	ISO 8601
`04:05`	ISO 8601
`040506`	ISO 8601

Пример	Описание
04:05 AM	то же, что и 04:05; AM не меняет значение времени
04:05 PM	то же, что и 16:05; часы должны быть <= 12
04:05:06.789-8	ISO 8601, с часовым поясом в виде смещения от UTC
04:05:06-08:00	ISO 8601, с часовым поясом в виде смещения от UTC
04:05-08:00	ISO 8601, с часовым поясом в виде смещения от UTC
040506-08	ISO 8601, с часовым поясом в виде смещения от UTC
040506+0730	ISO 8601, с часовым поясом, задаваемым нецелочисленным смещением от UTC
040506+07:30:00	смещение от UTC, заданное до секунд (не допускается в ISO 8601)
04:05:06 PST	часовой пояс задаётся аббревиатурой
2003-04-12 04:05:06 America/New_York	часовой пояс задаётся полным названием

Таблица 8.12. Вводимый часовой пояс

Пример	Описание
PST	аббревиатура (Pacific Standard Time, Стандартное тихоокеанское время)
America/New_York	полное название часового пояса
PST8PDT	указание часового пояса в стиле POSIX
-8:00:00	смещение часового пояса PST от UTC
-8:00	смещение часового пояса PST от UTC (расширенный формат ISO 8601)
-800	смещение часового пояса PST от UTC (стандартный формат ISO 8601)
-8	смещение часового пояса PST от UTC (стандартный формат ISO 8601)
zulu	принятое у военных сокращение UTC
z	краткая форма zulu (также определена в ISO 8601)

Подробнее узнать о том, как указывается часовой пояс, можно в Подразделе 8.5.3.

8.5.1.3. Даты и время

Допустимые значения типов timestamp состоят из записи даты и времени, после которого может указываться часовой пояс и необязательное уточнение AD или BC, определяющее эпоху до нашей эры и нашу эру соответственно. (AD/BC можно указать и перед часовым поясом, но предпочтительнее первый вариант.) Таким образом:

```
1999-01-08 04:05:06
```

и

```
1999-01-08 04:05:06 -8:00
```

допустимые варианты, соответствующие стандарту ISO 8601. В дополнение к этому поддерживается распространённый формат:

```
January 8 04:05:06 1999 PST
```

Стандарт SQL различает константы типов `timestamp without time zone` и `timestamp with time zone` по знаку «+» или «-» и смещению часового пояса, добавленному после времени. Следовательно, согласно стандарту, записи

```
TIMESTAMP '2004-10-19 10:23:54'
```

должен соответствовать тип `timestamp without time zone`, а

```
TIMESTAMP '2004-10-19 10:23:54+02'
```

тип `timestamp with time zone`. Postgres Pro никогда не анализирует содержимое текстовой строки, чтобы определить тип значения, и поэтому обе записи будут обработаны как значения типа `timestamp without time zone`. Чтобы текстовая константа обрабатывалась как `timestamp with time zone`, укажите этот тип явно:

```
TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'
```

В константе типа `timestamp without time zone` Postgres Pro просто игнорирует часовой пояс. То есть результирующее значение вычисляется только из полей даты/времени и не подстраивается под указанный часовой пояс.

Значения `timestamp with time zone` внутри всегда хранятся в UTC (Universal Coordinated Time, Всемирное скоординированное время или время по Гринвичу, GMT). Вводимое значение, в котором явно указан часовой пояс, переводится в UTC с учётом смещения данного часового пояса. Если во входной строке не указан часовой пояс, подразумевается часовой пояс, заданный системным параметром TimeZone и время так же пересчитывается в UTC со смещением `timezone`.

Когда значение `timestamp with time zone` выводится, оно всегда преобразуется из UTC в текущий часовой пояс `timezone` и отображается как локальное время. Чтобы получить время для другого часового пояса, нужно либо изменить `timezone`, либо воспользоваться конструкцией `AT TIME ZONE` (см. Подраздел 9.9.3).

В преобразованиях между `timestamp without time zone` и `timestamp with time zone` обычно предполагается, что значение `timestamp without time zone` содержит местное время (для часового пояса `timezone`). Другой часовой пояс для преобразования можно задать с помощью `AT TIME ZONE`.

8.5.1.4. Специальные значения

Postgres Pro для удобства поддерживает несколько специальных значений даты/времени, перечисленных в Таблице 8.13. Значения `infinity` и `-infinity` имеют особое представление в системе и они отображаются в том же виде, тогда как другие варианты при чтении преобразуются в значения даты/времени. (В частности, `now` и подобные строки преобразуются в актуальные значения времени в момент чтения.) Чтобы использовать эти значения в качестве констант в командах SQL, их нужно заключать в апострофы.

Таблица 8.13. Специальные значения даты/времени

Вводимая строка	Допустимые типы	Описание
`epoch`	`date`, `timestamp`	1970-01-01 00:00:00+00 (точка отсчёта времени в Unix)
`infinity`	`date`, `timestamp`	время после максимальной допустимой даты
`-infinity`	`date`, `timestamp`	время до минимальной допустимой даты
`now`	`date`, `time`, `timestamp`	время начала текущей транзакции
`today`	`date`, `timestamp`	время начала текущих суток (`00:00`)

Вводимая строка	Допустимые типы	Описание
tomorrow	date, timestamp	время начала следующих суток (00:00)
yesterday	date, timestamp	время начала предыдущих суток (00:00)
allballs	time	00:00:00.00 UTC

Для получения текущей даты/времени соответствующего типа можно также использовать следующие SQL-совместимые функции: CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME и LOCALTIMESTAMP. (См. Подраздел 9.9.4.) Заметьте, что во входных строках эти SQL-функции *не* распознаются.

> **Внимание**
>
> Входные значения now, today, tomorrow и yesterday вполне корректно работают в интерактивных SQL-командах, но когда команды сохраняются для последующего выполнения, например в подготовленных операторах, представлениях или определениях функций, их поведение может быть неожиданным. Такая строка может преобразоваться в конкретное значение времени, которое затем будет использоваться гораздо позже момента, когда оно было получено. В таких случаях следует использовать одну из SQL-функций. Например, CURRENT_DATE + 1 будет работать надёжнее, чем 'tomorrow'::date.

8.5.2. Вывод даты/времени

В качестве выходного формата типов даты/времени можно использовать один из четырёх стилей: ISO 8601, SQL (Ingres), традиционный формат POSTGRES (формат date в Unix) или German. По умолчанию выбран формат ISO. (Стандарт SQL требует, чтобы использовался именно ISO 8601. Другой формат называется «SQL» исключительно по историческим причинам.) Примеры всех стилей вывода перечислены в Таблице 8.14. Вообще со значениями типов date и time выводилась бы только часть даты или времени из показанных примеров, но со стилем POSTGRES значение даты без времени выводится в формате ISO.

Таблица 8.14. Стили вывода даты/время

Стиль	Описание	Пример
ISO	ISO 8601, стандарт SQL	1997-12-17 07:37:16-08
SQL	традиционный стиль	12/17/1997 07:37:16.00 PST
Postgres	изначальный стиль	Wed Dec 17 07:37:16 1997 PST
German	региональный стиль	17.12.1997 07:37:16.00 PST

> **Примечание**
>
> ISO 8601 указывает, что дата должна отделяться от времени буквой T в верхнем регистре. Postgres Pro принимает этот формат при вводе, но при выводе вставляет вместо T пробел, как показано выше. Это сделано для улучшения читаемости и для совместимости с RFC 3339 и другими СУБД.

В стилях SQL и POSTGRES день выводится перед месяцем, если установлен порядок DMY, а в противном случае месяц выводится перед днём. (Как этот параметр также влияет на интерпретацию входных значений, описано в Подразделе 8.5.1) Соответствующие примеры показаны в Таблице 8.15.

Таблица 8.15. Соглашения о порядке компонентов даты

Параметр `datestyle`	Порядок при вводе	Пример вывода
`SQL, DMY`	*день*/*месяц*/*год*	`17/12/1997 15:37:16.00 CET`
`SQL, MDY`	*месяц*/*день*/*год*	`12/17/1997 07:37:16.00 PST`
`Postgres, DMY`	*день*/*месяц*/*год*	`Wed 17 Dec 07:37:16 1997 PST`

В формате ISO часовой пояс всегда отображается в виде числового смещения со знаком относительно всемирного координированного времени (UTC); для зон к востоку от Гринвича знак смещения положительный. Смещение будет отображаться как *hh* (только часы), если оно задаётся целым числом часов, как *hh*:*mm*, если оно задаётся целым числом минут, или как *hh*:*mm*:*ss*. (Третий вариант невозможен в современных стандартах часовых поясов, но он может понадобиться при работе с отметками времени, предшествующими принятию стандартизированных часовых поясов.) В других форматах даты часовой пояс отображается как буквенное сокращение, если оно принято в текущем поясе. В противном случае он отображается как числовое смещение со знаком в стандартном формате ISO 8601 (*hh* или *hhmm*).

Стиль даты/времени пользователь может выбрать с помощью команды `SET datestyle`, параметра DateStyle в файле конфигурации `postgresql.conf` или переменной окружения `PGDATESTYLE` на сервере или клиенте.

Для большей гибкости при форматировании выводимой даты/времени можно использовать функцию `to_char` (см. Раздел 9.8).

8.5.3. Часовые пояса

Часовые пояса и правила их применения определяются, как вы знаете, не только по географическим, но и по политическим соображениям. Часовые пояса во всём мире были более-менее стандартизированы в начале прошлого века, но они продолжают претерпевать изменения, в частности это касается перехода на летнее время. Для расчёта времени в прошлом Postgres Pro получает исторические сведения о правилах часовых поясов из распространённой базы данных IANA (Olson). Для будущего времени предполагается, что в заданном часовом поясе будут продолжать действовать последние принятые правила.

Postgres Pro стремится к совместимости со стандартом SQL в наиболее типичных случаях. Однако стандарт SQL допускает некоторые странности при смешивании типов даты и времени. Две очевидные проблемы:

- Хотя для типа `date` часовой пояс указать нельзя, это можно сделать для типа `time`. В реальности это не очень полезно, так как без даты нельзя точно определить смещение при переходе на летнее время.
- По умолчанию часовой пояс задаётся постоянным смещением от UTC. Это также не позволяет учесть летнее время при арифметических операций с датами, пересекающими границы летнего времени.

Поэтому мы советуем использовать часовой пояс с типами, включающими и время, и дату. Мы *не* рекомендуем использовать тип `time with time zone` (хотя Postgres Pro поддерживает его для старых приложений и совместимости со стандартом SQL). Для типов, включающих только дату или только время, в Postgres Pro предполагается местный часовой пояс.

Все значения даты и времени с часовым поясом представляются внутри в UTC, а при передаче клиентскому приложению они переводятся в местное время, при этом часовой пояс по умолчанию определяется параметром конфигурации TimeZone.

Postgres Pro позволяет задать часовой пояс тремя способами:

- Полное название часового пояса, например `America/New_York`. Все допустимые названия перечислены в представлении `pg_timezone_names` (см. Раздел 49.90). Определения часовых поясов Postgres Pro берёт из широко распространённой базы IANA, так что имена часовых поясов Postgres Pro будут воспринимать и другие приложения.

- Аббревиатура часового пояса, например `PST`. Такое определение просто задаёт смещение от UTC, в отличие от полных названий поясов, которые кроме того подразумевают и правила перехода на летнее время. Распознаваемые аббревиатуры перечислены в представлении `pg_timezone_abbrevs` (см. Раздел 49.89). Аббревиатуры можно использовать во вводимых значениях даты/времени и в операторе `AT TIME ZONE`, но не в параметрах конфигурации TimeZone и log_timezone.
- Помимо аббревиатур и названий часовых поясов Postgres Pro принимает указания часовых поясов в стиле POSIX, как описано в Разделе B.5. Этот вариант обычно менее предпочтителен, чем использование именованного часового пояса, но он может быть единственным возможным, если для нужного часового пояса нет записи в базе данных IANA.

Вкратце, различие между аббревиатурами и полными названиями заключаются в следующем: аббревиатуры представляют определённый сдвиг от UTC, а полное название подразумевает ещё и местное правило по переходу на летнее время, то есть, возможно, два сдвига от UTC. Например, `2014-06-04 12:00 America/New_York` представляет полдень по местному времени в Нью-Йорк, что для данного дня было бы летним восточным временем (EDT или UTC-4). Так что `2014-06-04 12:00 EDT` обозначает тот же момент времени. Но `2014-06-04 12:00 EST` задаёт стандартное восточное время (UTC-5), не зависящее о того, действовало ли летнее время в этот день.

Мало того, в некоторых юрисдикциях одна и та же аббревиатура часового пояса означала разные сдвиги UTC в разное время; например, аббревиатура московского времени `MSK` несколько лет означала UTC+3, а затем стала означать UTC+4. Postgres Pro обрабатывает такие аббревиатуры в соответствии с их значениями на заданную дату, но, как и с примером выше `EST`, это не обязательно будет соответствовать местному гражданскому времени в этот день.

Независимо от формы, регистр в названиях и аббревиатурах часовых поясов не важен. (В PostgreSQL до версии 8.2 он где-то имел значение, а где-то нет.)

Ни названия, ни аббревиатуры часовых поясов, не зашиты в самом сервере; они считываются из файлов конфигурации, находящихся в путях `.../share/timezone/` и `.../share/timezonesets/` относительно каталога установки (см. Раздел B.4).

Параметр конфигурации TimeZone можно установить в `postgresql.conf` или любым другим стандартным способом, описанным в Главе 18. Часовой пояс может быть также определён следующими специальными способами:

- Часовой пояс для текущего сеанса можно установить с помощью SQL-команды `SET TIME ZONE`. Это альтернативная запись команды `SET TIMEZONE TO`, более соответствующая SQL-стандарту.
- Если установлена переменная окружения `PGTZ`, клиенты libpq используют её значение, выполняя при подключении к серверу команду `SET TIME ZONE`.

8.5.4. Ввод интервалов

Значения типа `interval` могут быть записаны в следующей расширенной форме:

`[@]` *количество единица* `[`*количество единица*`...] [`*направление*`]`

где *количество* — это число (возможно, со знаком); *единица* — одно из значений: `microsecond`, `millisecond`, `second`, `minute`, `hour`, `day`, `week`, `month`, `year`, `decade`, `century`, `millennium` (которые обозначают соответственно микросекунды, миллисекунды, секунды, минуты, часы, дни, недели, месяцы, годы, десятилетия, века и тысячелетия), либо эти же слова во множественном числе, либо их сокращения; *направление* может принимать значение `ago` (назад) или быть пустым. Знак `@` является необязательным. Все заданные величины различных единиц суммируются вместе с учётом знака чисел. Указание `ago` меняет знак всех полей на противоположный. Этот синтаксис также используется при выводе интервала, если параметр IntervalStyle имеет значение `postgres_verbose`.

Количества дней, часов, минут и секунд можно определить, не указывая явно соответствующие единицы. Например, запись `'1 12:59:10'` равнозначна `'1 day 12 hours 59 min 10 sec'`. Сочетание

года и месяца также можно записать через минус; например '200-10' означает то, же что и '200 years 10 months'. (На самом деле только эти краткие формы разрешены стандартом SQL и они используются при выводе, когда IntervalStyle имеет значение sql_standard.)

Интервалы можно также записывать в виде, определённом в ISO 8601, либо в «формате с кодами», описанном в разделе 4.4.3.2 этого стандарта, либо в «альтернативном формате», описанном в разделе 4.4.3.3. Формат с кодами выглядит так:

P *количество единица* [*количество единица* ...] [T [*количество единица* ...]]

Строка должна начинаться с символа P и может включать также T перед временем суток. Допустимые коды единиц перечислены в Таблице 8.16. Коды единиц можно опустить или указать в любом порядке, но компоненты времени суток должны идти после символа T. В частности, значение кода M зависит от того, располагается ли он до или после T.

Таблица 8.16. Коды единиц временных интервалов ISO 8601

Код	Значение
Y	годы
M	месяцы (в дате)
W	недели
D	дни
H	часы
M	минуты (во времени)
S	секунды

В альтернативном формате:

P [*год-месяц-день*] [T *часы*:*минуты*:*секунды*]

строка должна начинаться с P, а T разделяет компоненты даты и времени. Значения выражаются числами так же, как и в датах ISO 8601.

При записи интервальной константы с указанием *полей* или присваивании столбцу типа interval строки с *полями*, интерпретация непомеченных величин зависит от *полей*. Например, INTERVAL '1' YEAR воспринимается как 1 год, а INTERVAL '1' — как 1 секунда. Кроме того, значения «справа» от меньшего значащего поля, заданного в определении *полей*, просто отбрасываются. Например, в записи INTERVAL '1 day 2:03:04' HOUR TO MINUTE будут отброшены секунды, но не день.

Согласно стандарту SQL, все компоненты значения interval должны быть одного знака, и ведущий минус применяется ко всем компонентам; например, минус в записи '-1 2:03:04' применяется и к дню, и к часам/минутам/секундам. Postgres Pro позволяет задавать для разных компонентов разные знаки и традиционно обрабатывает знак каждого компонента в текстовом представлении отдельно от других, так что в данном случае часы/минуты/секунды будут считаться положительными. Если параметр IntervalStyle имеет значение sql_standard, ведущий знак применяется ко всем компонентам (но только если они не содержат знаки явно). В противном случае действуют традиционные правила Postgres Pro. Во избежание неоднозначности рекомендуется добавлять знак к каждому компоненту с отрицательным значением.

Значения полей могут содержать дробную часть: например, '1.5 weeks' или '01: 02: 03.45'. Однако поскольку интервал содержит только три целых единицы (месяцы, дни, микросекунды), дробные единицы должны быть разделены на более мелкие единицы. Дробные части единиц, превышающих месяцы, усекаются до целого числа месяцев, например '1.5 years' преобразуется в '1 year 6 mons'. Дробные части недель и дней пересчитываются в целое число дней и микросекунд, из расчёта, что в месяце 30 дней, а в сутках — 24 часа, например, '1.75 months' становится 1 mon 22 days 12:00:00. На выходе только секунды могут иметь дробную часть.

В Таблице 8.17 показано несколько примеров допустимых вводимых значений типа interval.

Таблица 8.17. Ввод интервалов

Пример	Описание
1-2	Стандартный формат SQL: 1 год и 2 месяца
3 4:05:06	Стандартный формат SQL: 3 дня 4 часа 5 минут 6 секунд
1 year 2 months 3 days 4 hours 5 minutes 6 seconds	Традиционный формат Postgres: 1 год 2 месяца 3 дня 4 часа 5 минут 6 секунд
P1Y2M3DT4H5M6S	«Формат с кодами» ISO 8601: то же значение, что и выше
P0001-02-03T04:05:06	«Альтернативный формат» ISO 8601: то же значение, что и выше

Тип `interval` представлен внутри в виде отдельных значений месяцев, дней и микросекунд. Это объясняется тем, что число дней в месяце может быть разным, а в сутках может быть и 23, и 25 часов в дни перехода на летнее/зимнее время. Значения месяцев и дней представлены целыми числами, а в микросекундах могут представляться секунды с дробной частью. Так как интервалы обычно получаются из строковых констант или при вычитании типов `timestamp`, этот способ хранения эффективен в большинстве случаев, но может давать неожиданные результаты:

```
SELECT EXTRACT(hours from '80 minutes'::interval);
 date_part
-----------
         1

SELECT EXTRACT(days from '80 hours'::interval);
 date_part
-----------
         0
```

Для корректировки числа дней и часов, когда они выходят за обычные границы, есть специальные функции `justify_days` и `justify_hours`.

8.5.5. Вывод интервалов

Формат вывода типа interval может определяться одним из четырёх стилей: `sql_standard`, `postgres`, `postgres_verbose` и `iso_8601`. Выбрать нужный стиль позволяет команда `SET intervalstyle` (по умолчанию выбран `postgres`). Примеры форматов разных стилей показаны в Таблице 8.18.

Стиль `sql_standard` выдаёт результат, соответствующий стандарту SQL, если значение интервала удовлетворяет ограничениям стандарта (и содержит либо только год и месяц, либо только день и время, и при этом все его компоненты одного знака). В противном случае выводится год-месяц, за которым идёт дата-время, а в компоненты для однозначности явно добавляются знаки.

Вывод в стиле `postgres` соответствует формату, который был принят в PostgreSQL до версии 8.4, когда параметр DateStyle имел значение `ISO`.

Вывод в стиле `postgres_verbose` соответствует формату, который был принят в PostgreSQL до версии 8.4, когда значением параметром `DateStyle` было не `ISO`.

Вывод в стиле `iso_8601` соответствует «формату с кодами» описанному в разделе 4.4.3.2 формата ISO 8601.

Таблица 8.18. Примеры стилей вывода интервалов

Стиль	Интервал год-месяц	Интервал день-время	Смешанный интервал
`sql_standard`	1-2	3 4:05:06	-1-2 +3 -4:05:06

Стиль	Интервал год-месяц	Интервал день-время	Смешанный интервал
postgres	1 year 2 mons	3 days 04:05:06	-1 year -2 mons +3 days -04:05:06
postgres_verbose	@ 1 year 2 mons	@ 3 days 4 hours 5 mins 6 secs	@ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago
iso_8601	P1Y2M	P3DT4H5M6S	P-1Y-2M3DT-4H-5M-6S

8.6. Логический тип

В Postgres Pro есть стандартный SQL-тип boolean; см. Таблицу 8.19. Тип boolean может иметь следующие состояния: «true», «false» и третье состояние, «unknown», которое представляется SQL-значением NULL.

Таблица 8.19. Логический тип данных

Имя	Размер	Описание
boolean	1 байт	состояние: истина или ложь

Логические константы могут представляться в SQL-запросах следующими ключевыми словами SQL: TRUE, FALSE и NULL.

Функция ввода данных типа boolean воспринимает следующие строковые представления состояния «true»:

```
true
yes
on
1
```

и следующие представления состояния «false»:

```
false
no
off
0
```

Также воспринимаются уникальные префиксы этих строк, например t или n. Регистр символов не имеет значения, а пробельные символы в начале и в конце строки игнорируются.

Функция вывода данных типа boolean всегда выдаёт t или f, как показано в Примере 8.2.

Пример 8.2. Использование типа boolean

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
 a |    b
---+---------
 t | sic est
 f | non est

SELECT * FROM test1 WHERE a;
 a |    b
---+---------
 t | sic est
```

Ключевые слова TRUE и FALSE являются предпочтительными (соответствующими стандарту SQL) для записи логических констант в SQL-запросах. Но вы также можете использовать

строковые представления, которые допускает синтаксис строковых констант, описанный в Подразделе 4.1.2.7, например, `'yes'::boolean`.

Заметьте, что при анализе запроса `TRUE` и `FALSE` автоматически считаются значениями типа `boolean`, но для `NULL` это не так, потому что ему может соответствовать любой тип. Поэтому в некоторых контекстах может потребоваться привести `NULL` к типу `boolean` явно, например так: `NULL::boolean`. С другой стороны, приведение строковой константы к логическому типу можно опустить в тех контекстах, где анализатор запроса может понять, что буквальное значение должно иметь тип `boolean`.

8.7. Типы перечислений

Типы перечислений (enum) определяют статический упорядоченный набор значений, так же как и типы `enum`, существующие в ряде языков программирования. В качестве перечисления можно привести дни недели или набор состояний.

8.7.1. Объявление перечислений

Тип перечислений создаются с помощью команды CREATE TYPE, например так:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

Созданные типы enum можно использовать в определениях таблиц и функций, как и любые другие:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TABLE person (
    name text,
    current_mood mood
);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
 name | current_mood
------+--------------
 Moe  | happy
(1 row)
```

8.7.2. Порядок

Порядок значений в перечислении определяется последовательностью, в которой были указаны значения при создании типа. Перечисления поддерживаются всеми стандартными операторами сравнения и связанными агрегатными функциями. Например:

```
INSERT INTO person VALUES ('Larry', 'sad');
INSERT INTO person VALUES ('Curly', 'ok');
SELECT * FROM person WHERE current_mood > 'sad';
 name  | current_mood
-------+--------------
 Moe   | happy
 Curly | ok
(2 rows)

SELECT * FROM person WHERE current_mood > 'sad' ORDER BY current_mood;
 name  | current_mood
-------+--------------
 Curly | ok
 Moe   | happy
(2 rows)

SELECT name
FROM person
WHERE current_mood = (SELECT MIN(current_mood) FROM person);
```

```
 name
-------
 Larry
(1 row)
```

8.7.3. Безопасность типа

Все типы перечислений считаются уникальными и поэтому значения разных типов нельзя сравнивать. Взгляните на этот пример:

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');
CREATE TABLE holidays (
    num_weeks integer,
    happiness happiness
);
INSERT INTO holidays(num_weeks,happiness) VALUES (4, 'happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (6, 'very happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (8, 'ecstatic');
INSERT INTO holidays(num_weeks,happiness) VALUES (2, 'sad');
ОШИБКА:  неверное значение для перечисления happiness: "sad"
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood = holidays.happiness;
ОШИБКА:  оператор не существует: mood = happiness
```

Если вам действительно нужно сделать что-то подобное, вы можете либо реализовать собственный оператор, либо явно преобразовать типы в запросе:

```
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood::text = holidays.happiness::text;
 name | num_weeks
------+-----------
 Moe  |         4
(1 row)
```

8.7.4. Тонкости реализации

В метках значений регистр имеет значение, т. е. `'happy'` и `'HAPPY'` — не одно и то же. Также в метках имеют значение пробелы.

Хотя типы-перечисления предназначены прежде всего для статических наборов значений, имеется возможность добавлять новые значения в существующий тип-перечисление и переименовывать значения (см. ALTER TYPE). Однако удалять существующие значения из перечисления, а также изменять их порядок, нельзя — для получения нужного результата придётся удалить и воссоздать это перечисление.

Значение enum занимает на диске 4 байта. Длина текстовой метки значения ограничена параметром компиляции `NAMEDATALEN`; в стандартных сборках Postgres Pro он ограничивает длину 63 байтами.

Сопоставления внутренних значений enum с текстовыми метками хранятся в системном каталоге `pg_enum`. Он может быть полезен в ряде случаев.

8.8. Геометрические типы

Геометрические типы данных представляют объекты в двумерном пространстве. Все существующие в Postgres Pro геометрические типы перечислены в Таблице 8.20.

Таблица 8.20. Геометрические типы

Имя	Размер	Описание	Представление
`point`	16 байт	Точка на плоскости	(x,y)

Имя	Размер	Описание	Представление
line	32 байта	Бесконечная прямая	{A,B,C}
lseg	32 байта	Отрезок	((x1,y1),(x2,y2))
box	32 байта	Прямоугольник	((x1,y1),(x2,y2))
path	16+16n байт	Закрытый путь (подобный многоугольнику)	((x1,y1),...)
path	16+16n байт	Открытый путь	[(x1,y1),...]
polygon	40+16n байт	Многоугольник (подобный закрытому пути)	((x1,y1),...)
circle	24 байта	Окружность	<(x,y),r> (центр окружности и радиус)

Для выполнения различных геометрических операций, в частности масштабирования, вращения и определения пересечений, PostgreSQL предлагает богатый набор функций и операторов. Они рассматриваются в Разделе 9.11.

8.8.1. Точки

Точки — это основной элемент, на базе которого строятся все остальные геометрические типы. Значения типа point записываются в одном из двух форматов:

```
( x , y )
  x , y
```

где *x* и *y* — координаты точки на плоскости, выраженные числами с плавающей точкой.

Выводятся точки в первом формате.

8.8.2. Прямые

Прямые представляются линейным уравнением *A*x + *B*y + *C* = 0, где *A* и *B* не могут быть равны 0 одновременно. Значения типа line вводятся и выводятся в следующем виде:

```
{ A, B, C }
```

Кроме того, для ввода может использоваться любая из этих форм:

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

где (*x1*,*y1*) и (*x2*,*y2*) — две различные точки на данной прямой.

8.8.3. Отрезки

Отрезок представляется парой точек, определяющих концы отрезка. Значения типа lseg записываются в одной из следующих форм:

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

где (*x1*,*y1*) и (*x2*,*y2*) — концы отрезка.

Выводятся отрезки в первом формате.

8.8.4. Прямоугольники

Прямоугольник представляется двумя точками, находящимися в противоположных его углах. Значения типа `box` записываются в одной из следующих форм:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

где `(x1,y1)` и `(x2,y2)` — противоположные углы прямоугольника.

Выводятся прямоугольники во второй форме.

Во вводимом значении могут быть указаны любые два противоположных угла, но затем они будут упорядочены, так что внутри сохранятся правый верхний и левый нижний углы, в таком порядке.

8.8.5. Пути

Пути представляют собой списки соединённых точек. Пути могут быть *закрытыми*, когда подразумевается, что первая и последняя точка в списке соединены, или *открытыми*, в противном случае.

Значения типа `path` записываются в одной из следующих форм:

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

где точки задают узлы сегментов, составляющих путь. Квадратные скобки (`[]`) указывают, что путь открытый, а круглые (`()`) — закрытый. Когда внешние скобки опускаются, как в показанных выше последних трёх формах, считается, что путь закрытый.

Пути выводятся в первой или второй форме, в соответствии с типом.

8.8.6. Многоугольники

Многоугольники представляются списками точек (вершин). Многоугольники похожи на закрытые пути, но хранятся в другом виде и для работы с ними предназначен отдельный набор функций.

Значения типа `polygon` записываются в одной из следующих форм:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

где точки задают узлы сегментов, образующих границу многоугольника.

Выводятся многоугольники в первом формате.

8.8.7. Окружности

Окружности задаются координатами центра и радиусом. Значения типа `circle` записываются в одном из следующих форматов:

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

где `(x,y)` — центр окружности, а *r* — её радиус.

Выводятся окружности в первом формате.

8.9. Типы, описывающие сетевые адреса

Postgres Pro предлагает типы данных для хранения адресов IPv4, IPv6 и MAC, показанные в Таблице 8.21. Для хранения сетевых адресов лучше использовать эти типы, а не простые текстовые строки, так как Postgres Pro проверяет вводимые значения данных типов и предоставляет специализированные операторы и функции для работы с ними (см. Раздел 9.12).

Таблица 8.21. Типы, описывающие сетевые адреса

Имя	Размер	Описание
`cidr`	7 или 19 байт	Сети IPv4 и IPv6
`inet`	7 или 19 байт	Узлы и сети IPv4 и IPv6
`macaddr`	6 байт	MAC-адреса
`macaddr8`	8 байт	MAC-адреса (в формате EUI-64)

При сортировке типов `inet` и `cidr`, адреса IPv4 всегда идут до адресов IPv6, в том числе адреса IPv4, включённые в IPv6 или сопоставленные с ними, например ::10.2.3.4 или ::ffff:10.4.3.2.

8.9.1. `inet`

Тип `inet` содержит IPv4- или IPv6-адрес узла и может также содержать его подсеть, всё в одном поле. Подсеть представляется числом бит, определяющих адрес сети в адресе узла (или «маску сети»). Если маска сети равна 32 для адреса IPv4, такое значение представляет не подсеть, а определённый узел. Адреса IPv6 имеют длину 128 бит, поэтому уникальный адрес узла задаётся с маской 128 бит. Заметьте, что когда нужно, чтобы принимались только адреса сетей, следует использовать тип `cidr`, а не `inet`.

Вводимые значения такого типа должны иметь формат *`IP-адрес/y`*, где *`IP-адрес`* — адрес IPv4 или IPv6, а *`y`* — число бит в маске сети. Если компонент *`/y`* отсутствует, маска сети считается равной 32 для IPv4 и 128 для IPv6, так что это значение будет представлять один узел. При выводе компонент *`/y`* опускается, если сетевой адрес определяет адрес одного узла.

8.9.2. `cidr`

Тип `cidr` содержит определение сети IPv4 или IPv6. Входные и выходные форматы соответствуют соглашениям CIDR (Classless Internet Domain Routing, Бесклассовая межсетевая адресация). Определение сети записывается в формате *`IP-адрес/y`*, где *`IP-адрес`* — адрес сети IPv4 или IPv6, а *`y`* — число бит в маске сети. Если *`y`* не указывается, это значение вычисляется по старой классовой схеме нумерации сетей, но при этом оно может быть увеличено, чтобы в него вошли все байты введённого адреса. Если в сетевом адресе справа от маски сети окажутся биты со значением 1, он будет считаться ошибочным.

В Таблице 8.22 показаны несколько примеров адресов.

Таблица 8.22. Примеры допустимых значений типа `cidr`

Вводимое значение `cidr`	Выводимое значение `cidr`	`abbrev( cidr)`
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16

Вводимое значение `cidr`	**Выводимое значение `cidr`**	`abbrev( cidr)`
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba::/64
2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

8.9.3. Различия `inet` и `cidr`

Существенным различием типов данных `inet` и `cidr` является то, что `inet` принимает значения с ненулевыми битами справа от маски сети, а `cidr` — нет. Например, значение `192.168.0.1/24` является допустимым для типа `inet`, но не для `cidr`.

> **Подсказка**
>
> Если вас не устраивает выходной формат значений `inet` или `cidr`, попробуйте функции `host`, `text` и `abbrev`.

8.9.4. `macaddr`

Тип `macaddr` предназначен для хранения MAC-адреса, примером которого является адрес сетевой платы Ethernet (хотя MAC-адреса применяются и для других целей). Вводимые значения могут задаваться в следующих форматах:

```
'08:00:2b:01:02:03'
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'0800-2b01-0203'
'08002b010203'
```

Все эти примеры определяют один и тот же адрес. Шестнадцатеричные цифры от `a` до `f` могут быть и в нижнем, и в верхнем регистре. Выводятся MAC-адреса всегда в первой форме.

Стандарт IEEE 802-2001 считает канонической формой MAC-адресов вторую (с минусами), а в первой (с двоеточиями) предполагает обратный порядок бит, от старшего к младшему, так что 08-00-2b-01-02-03 = 10:00:D4:80:40:C0. В настоящее время этому соглашению практически никто не следует, и уместно оно было только для устаревших сетевых протоколов (таких как Token Ring). Postgres Pro не меняет порядок бит и во всех принимаемых форматах подразумевается традиционный порядок LSB.

Последние пять входных форматов не описаны ни в каком стандарте.

8.9.5. `macaddr8`

Тип `macaddr8` хранит MAC-адреса в формате EUI-64, применяющиеся, например, для аппаратных адресов плат Ethernet (хотя MAC-адреса используются и для других целей). Этот тип может принять и 6-байтовые, и 8-байтовые адреса MAC и сохраняет их в 8 байтах. MAC-адреса,

заданные в 6-байтовом формате, хранятся в формате 8 байт, а 4-ый и 5-ый байт содержат FF и FE, соответственно. Заметьте, что для IPv6 используется модифицированный формат EUI-64, в котором 7-ой бит должен быть установлен в 1 после преобразования из EUI-48. Для выполнения этого изменения предоставляется функция `macaddr8_set7bit`. Вообще говоря, этот тип принимает любые строки, состоящие из пар шестнадцатеричных цифр (выровненных по границам байт), которые могут согласованно разделяться одинаковыми символами `':'`, `'-'` или `'.'`. Шестнадцатеричных цифр должно быть либо 16 (8 байт), либо 12 (6 байт). Ведущие и замыкающие пробелы игнорируются. Ниже показаны примеры допустимых входных строк:

```
'08:00:2b:01:02:03:04:05'
'08-00-2b-01-02-03-04-05'
'08002b:0102030405'
'08002b-0102030405'
'0800.2b01.0203.0405'
'0800-2b01-0203-0405'
'08002b01:02030405'
'08002b0102030405'
```

Во всех этих примерах задаётся один и тот же адрес. Для цифр с `a` по `f` принимаются буквы и в верхнем, и в нижнем регистре. Вывод всегда представляется в первом из показанных форматов. Последние шесть входных форматов, показанных выше, не являются стандартизированными. Чтобы преобразовать традиционный 48-битный MAC-адрес в формате EUI-48 в модифицированный формат EUI-64 для включения в состав адреса IPv6 в качестве адреса узла, используйте функцию `macaddr8_set7bit`:

```
SELECT macaddr8_set7bit('08:00:2b:01:02:03');

    macaddr8_set7bit
-------------------------
 0a:00:2b:ff:fe:01:02:03
(1 row)
```

8.10. Битовые строки

Битовые строки представляют собой последовательности из 1 и 0. Их можно использовать для хранения или отображения битовых масок. В SQL есть два битовых типа: `bit(`*n*`)` и `bit varying(`*n*`)`, где *n* — положительное целое число.

Длина значения типа `bit` должна в точности равняться *n*; при попытке сохранить данные длиннее или короче произойдёт ошибка. Данные типа `bit varying` могут иметь переменную длину, но не превышающую *n*; строки большей длины не будут приняты. Запись `bit` без указания длины равнозначна записи `bit(1)`, тогда как `bit varying` без указания длины подразумевает строку неограниченной длины.

> **Примечание**
>
> При попытке привести значение битовой строки к типу `bit(`*n*`)`, оно будет усечено или дополнено нулями справа до длины ровно *n* бит, ошибки при этом не будет. Подобным образом, если явно привести значение битовой строки к типу `bit varying(`*n*`)`, она будет усечена справа, если её длина превышает *n* бит.

Синтаксис констант битовых строк описан в Подразделе 4.1.2.5, а все доступные битовые операторы и функции перечислены в Разделе 9.6.

Пример 8.3. Использование битовых строк

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
```

```
INSERT INTO test VALUES (B'10', B'101');

ОШИБКА:  длина битовой строки (2) не соответствует типу bit(3)

INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;

  a  |  b
-----+-----
 101 | 00
 100 | 101
```

Для хранения битовой строки используется по 1 байту для каждой группы из 8 бит, плюс 5 или 8 байт дополнительно в зависимости от длины строки (но длинные строки могут быть сжаты или вынесены отдельно, как описано в Разделе 8.3 применительно к символьным строкам).

8.11. Типы, предназначенные для текстового поиска

Postgres Pro предоставляет два типа данных для поддержки полнотекстового поиска. Текстовым поиском называется операция анализа набора *документов* с текстом на естественном языке, в результате которой находятся фрагменты, наиболее соответствующие *запросу*. Тип `tsvector` представляет документ в виде, оптимизированном для текстового поиска, а `tsquery` представляет запрос текстового поиска в подобном виде. Более подробно это описывается в Главе 12, а все связанные функции и операторы перечислены в Разделе 9.13.

8.11.1. `tsvector`

Значение типа `tsvector` содержит отсортированный список неповторяющихся *лексем*, т. е. слов, *нормализованных* так, что все словоформы сводятся к одной (подробнее это описано в Главе 12). Сортировка и исключение повторяющихся слов производится автоматически при вводе значения, как показано в этом примере:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
                      tsvector
----------------------------------------------------
 'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
```

Для представления в виде лексем пробелов или знаков препинания их нужно заключить в апострофы:

```
SELECT $$the lexeme '    ' contains spaces$$::tsvector;
                 tsvector
-------------------------------------------
 '    ' 'contains' 'lexeme' 'spaces' 'the'
```

(В данном и следующих примерах мы используем строку в долларах, чтобы не дублировать все апострофы в таких строках.) При этом включаемый апостроф или обратную косую черту нужно продублировать:

```
SELECT $$the lexeme 'Joe''s' contains a quote$$::tsvector;
                    tsvector
-------------------------------------------------
 'Joe''s' 'a' 'contains' 'lexeme' 'quote' 'the'
```

Также для лексем можно указать их целочисленные *позиции*:

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11
  rat:12'::tsvector;
                                  tsvector
-------------------------------------------------------------------------------
 'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12
  'sat':4
```

Позиция обычно указывает положение исходного слова в документе. Информация о расположении слов затем может использоваться для *оценки близости*. Позиция может задаваться числом от 1 до 16383; большие значения просто заменяются на 16383. Если для одной лексемы дважды указывается одно положение, такое повторение отбрасывается.

Лексемам, для которых заданы позиции, также можно назначить *вес*, выраженный буквами `A`, `B`, `C` или `D`. Вес `D` подразумевается по умолчанию и поэтому он не показывается при выводе:

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
          tsvector
----------------------------
 'a':1A 'cat':5 'fat':2B,4C
```

Веса обычно применяются для отражения структуры документа, например для придания особого значения словам в заголовке по сравнению со словами в обычном тексте. Назначенным весам можно сопоставить числовые приоритеты в функциях ранжирования результатов.

Важно понимать, что тип `tsvector` сам по себе не выполняет нормализацию слов; предполагается, что в сохраняемом значении слова уже нормализованы приложением. Например:

```
SELECT 'The Fat Rats'::tsvector;
      tsvector
--------------------
 'Fat' 'Rats' 'The'
```

Для большинства англоязычных приложений приведённые выше слова будут считаться ненормализованными, но для `tsvector` это не важно. Поэтому исходный документ обычно следует обработать функцией `to_tsvector`, нормализующей слова для поиска:

```
SELECT to_tsvector('english', 'The Fat Rats');
   to_tsvector
-----------------
 'fat':2 'rat':3
```

И это подробнее описано в Главе 12.

8.11.2. `tsquery`

Значение `tsquery` содержит искомые лексемы, объединяемые логическими операторами `&` (И), `|` (ИЛИ) и `!` (НЕ), а также оператором поиска фраз `<->` (ПРЕДШЕСТВУЕТ). Также допускается вариация оператора ПРЕДШЕСТВУЕТ вида `<`*`N`*`>`, где *`N`* — целочисленная константа, задающая расстояние между двумя искомыми лексемами. Запись оператора `<->` равнозначна `<1>`.

Для группировки операторов могут использоваться скобки. Без скобок эти операторы имеют разные приоритеты, в порядке убывания: `!` (НЕ), `<->` (ПРЕДШЕСТВУЕТ), `&` (И) и `|` (ИЛИ).

Несколько примеров:

```
SELECT 'fat & rat'::tsquery;
    tsquery
---------------
 'fat' & 'rat'

SELECT 'fat & (rat | cat)'::tsquery;
          tsquery
---------------------------
 'fat' & ( 'rat' | 'cat' )

SELECT 'fat & rat & ! cat'::tsquery;
        tsquery
------------------------
 'fat' & 'rat' & !'cat'
```

Лексемам в tsquery можно дополнительно сопоставить буквы весов, при этом они будут соответствовать только тем лексемам в tsvector, которые имеют какой-либо из этих весов:

```
SELECT 'fat:ab & cat'::tsquery;
    tsquery
------------------
 'fat':AB & 'cat'
```

Кроме того, в лексемах tsquery можно использовать знак * для поиска по префиксу:

```
SELECT 'super:*'::tsquery;
  tsquery
-----------
 'super':*
```

Этот запрос найдёт все слова в tsvector, начинающиеся с приставки «super».

Апострофы в лексемах этого типа можно использовать так же, как и в лексемах в tsvector; и так же, как и для типа tsvector, необходимая нормализация слова должна выполняться до приведения значения к типу tsquery. Для такой нормализации удобно использовать функцию to_tsquery:

```
SELECT to_tsquery('Fat:ab & Cats');
    to_tsquery
------------------
 'fat':AB & 'cat'
```

Заметьте, что функция to_tsquery будет обрабатывать префиксы подобно другим словам, поэтому следующее сравнение возвращает true:

```
SELECT to_tsvector( 'postgraduate' ) @@ to_tsquery( 'postgres:*' );
 ?column?
----------
 t
```

так как postgres преобразуется стеммером в postgr:

```
SELECT to_tsvector( 'postgraduate' ), to_tsquery( 'postgres:*' );
  to_tsvector  | to_tsquery
---------------+------------
 'postgradu':1 | 'postgr':*
```

и эта приставка находится в преобразованной форме слова postgraduate.

8.12. Тип UUID

Тип данных uuid сохраняет универсальные уникальные идентификаторы (Universally Unique Identifiers, UUID), определённые в RFC 4122, ISO/IEC 9834-8:2005 и связанных стандартах. (В некоторых системах это называется GUID, глобальным уникальным идентификатором.) Этот идентификатор представляет собой 128-битное значение, генерируемое специальным алгоритмом, практически гарантирующим, что этим же алгоритмом оно не будет получено больше нигде в мире. Таким образом, эти идентификаторы будут уникальными и в распределённых системах, а не только в единственной базе данных, как значения генераторов последовательностей.

UUID записывается в виде последовательности шестнадцатеричных цифр в нижнем регистре, разделённых знаками минуса на несколько групп, в таком порядке: группа из 8 цифр, за ней три группы из 4 цифр и, наконец, группа из 12 цифр, что в сумме составляет 32 цифры и представляет 128 бит. Пример UUID в этом стандартном виде:

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

Postgres Pro также принимает альтернативные варианты: цифры в верхнем регистре, стандартную запись, заключённую в фигурные скобки, запись без минусов или с минусами, разделяющими любые группы из четырёх цифр. Например:

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
a0eebc999c0b4ef8bb6d6bb9bd380a11
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}
```

Выводится значение этого типа всегда в стандартном виде.

В Postgres Pro встроены функции хранения и сравнения идентификаторов UUID, но нет внутренней функции генерирования UUID, потому что не существует какого-то единственного алгоритма, подходящего для всех приложений. Сгенерировать UUID можно с помощью дополнительного модуля uuid-ossp, в котором реализованы несколько стандартных алгоритмов, а можно воспользоваться модулем pgcrypto, где тоже есть функция генерирования случайных UUID. Кроме того, можно сделать это в клиентском приложении или в другой библиотеке, подключённой на стороне сервера.

8.13. Тип XML

Тип `xml` предназначен для хранения XML-данных. Его преимущество по сравнению с обычным типом `text` в том, что он проверяет вводимые значения на допустимость по правилам XML и для работы с ним есть типобезопасные функции; см. Раздел 9.14. Для использования этого типа дистрибутив должен быть скомпилирован в конфигурации `configure --with-libxml`.

Тип `xml` может сохранять правильно оформленные «документы», в соответствии со стандартом XML, а также фрагменты «содержимого», определяемые как менее ограниченные *«узлы документа»* в модели данных XQuery и XPath. Другими словами, это означает, что во фрагментах содержимого может быть несколько элементов верхнего уровня или текстовых узлов. Является ли некоторое значение типа `xml` полным документом или фрагментом содержимого, позволяет определить выражение *`xml-значение`* `IS DOCUMENT`.

Информацию о совместимости и ограничениях типа данных `xml` можно найти в Разделе D.3.

8.13.1. Создание XML-значений

Чтобы получить значение типа `xml` из текстовой строки, используйте функцию `xmlparse`:

```
XMLPARSE ( { DOCUMENT | CONTENT } value)
```

Примеры:

```
XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</title><chapter>...</
chapter></book>')
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

Хотя в стандарте SQL описан только один способ преобразования текстовых строк в XML-значения, специфический синтаксис Postgres Pro:

```
xml '<foo>bar</foo>'
'<foo>bar</foo>'::xml
```

тоже допустим.

Тип `xml` не проверяет вводимые значения по схеме DTD (Document Type Declaration, Объявления типа документа), даже если в них присутствуют ссылка на DTD. В настоящее время в PostgreSQL также нет встроенной поддержки других разновидностей схем, например XML Schema.

Обратная операция, получение текстовой строки из `xml`, выполняется с помощью функции `xmlserialize`:

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } значение AS тип )
```

Здесь допустимый *`тип`* — `character`, `character varying` или `text` (или их псевдонимы). И в данном случае стандарт SQL предусматривает только один способ преобразования `xml` в тип текстовых строк, но Postgres Pro позволяет просто привести значение к нужному типу.

При преобразовании текстовой строки в тип `xml` или наоборот без использования функций `XMLPARSE` и `XMLSERIALIZE`, выбор режима `DOCUMENT` или `CONTENT` определяется параметром конфигурации сеанса «XML option», установить который можно следующей стандартной командой:

```
SET XML OPTION { DOCUMENT | CONTENT };
```

или такой командой в духе Postgres Pro:

```
SET xmloption TO { DOCUMENT | CONTENT };
```

По умолчанию этот параметр имеет значение `CONTENT`, так что допускаются все формы XML-данных.

8.13.2. Обработка кодировки

Если на стороне сервера и клиента и в XML-данных используются разные кодировки символов, с этим могут возникать проблемы. Когда запросы передаются на сервер, а их результаты возвращаются клиенту в обычном текстовом режиме, Postgres Pro преобразует все передаваемые текстовые данные в кодировку для соответствующей стороны; см. Раздел 22.3. В том числе это происходит и со строковыми представлениями XML-данных, подобными тем, что показаны в предыдущих примерах. Обычно это означает, что объявления кодировки, содержащиеся в XML-данных, могут не соответствовать действительности, когда текстовая строка преобразуется из одной кодировки в другую при передаче данных между клиентом и сервером, так как подобные включённые в данные объявления не будут изменены автоматически. Для решения этой проблемы объявления кодировки, содержащиеся в текстовых строках, вводимых в тип `xml`, просто *игнорируются* и предполагается, что XML-содержимое представлено в текущей кодировке сервера. Как следствие, для правильной обработки таких строк с XML-данными клиент должен передавать их в своей текущей кодировке. Для сервера не важно, будет ли клиент для этого преобразовывать документы в свою кодировку, или изменит её, прежде чем передавать ему данные. При выводе значения типа `xml` не содержат объявления кодировки, а клиент должен предполагать, что все данные поступают в его текущей кодировке.

Если параметры запроса передаются на сервер и он возвращает результаты клиенту в двоичном режиме, кодировка символов не преобразуется, так что возникает другая ситуация. В этом случае объявление кодировки в XML принимается во внимание, а если его нет, то предполагается, что данные закодированы в UTF-8 (это соответствует стандарту XML; заметьте, что Postgres Pro не поддерживает UTF-16). При выводе в данные будет добавлено объявление кодировки, выбранной на стороне клиента (но если это UTF-8, объявление будет опущено).

Само собой, XML-данные в Postgres Pro будут обрабатываться гораздо эффективнее, когда и в XML-данных, и на стороне клиента, и на стороне сервера используется одна кодировка. Так как внутри XML-данные представляются в UTF-8, оптимальный вариант, когда на сервере также выбрана кодировка UTF-8.

> **Внимание**
>
> Некоторые XML-функции способны работать исключительно с ASCII-данными, если кодировка сервера не UTF-8. В частности, это известная особенность функций `xmltable()` и `xpath()`.

8.13.3. Обращение к XML-значениям

Тип `xml` отличается от других тем, что для него не определены никакие операторы сравнения, так как чётко определённого и универсального алгоритма сравнения XML-данных не существует. Одно из следствий этого — нельзя отфильтровать строки таблицы, сравнив столбец `xml` с искомым значением. Поэтому обычно XML-значения должны дополняться отдельным ключевым полем, например ID. Можно также сравнивать XML-значения, преобразовав их сначала в текстовые строки, но заметьте, что с учётом специфики XML-данных этот метод практически бесполезен.

Из-за отсутствия операторов сравнения для типа `xml`, для столбца этого типа также нельзя создать индекс. Поэтому, когда требуется быстрый поиск в XML данных, обойти это ограничение можно, приведя данные к типу текстовой строки и проиндексировав эти строки, либо проиндексировав выражение XPath. Конечно сам запрос при этом следует изменить, чтобы поиск выполнялся по индексированному выражению.

Для ускорения поиска в XML-данных также можно использовать функции полнотекстового поиска в Postgres Pro. Однако это требует определённой подготовки данных, что дистрибутив Postgres Pro пока не поддерживает.

8.14. Типы JSON

Типы JSON предназначены для хранения данных JSON (JavaScript Object Notation, Запись объекта JavaScript) согласно стандарту *RFC 7159*. Такие данные можно хранить и в типе `text`, но типы JSON лучше тем, что проверяют, соответствует ли вводимое значение формату JSON. Для работы с ними есть также несколько специальных функций и операторов; см. Раздел 9.15.

Существуют два типа данных JSON: `json` и `jsonb`. Они принимают на вход *почти* одинаковые наборы значений, а отличаются главным образом с точки зрения эффективности. Тип `json` сохраняет точную копию введённого текста, которую функции обработки должны разбирать заново при каждом выполнении запроса, тогда как данные `jsonb` сохраняются в разобранном двоичном формате, что несколько замедляет ввод из-за преобразования, но значительно ускоряет обработку, не требуя многократного разбора текста. Кроме того, `jsonb` поддерживает индексацию, что тоже может быть очень полезно.

Так как тип `json` сохраняет точную копию введённого текста, он сохраняет семантически незначащие пробелы между элементами, а также порядок ключей в JSON-объектах. И если JSON-объект внутри содержит повторяющиеся ключи, этот тип сохранит все пары ключ/значение. (Функции обработки будут считать действительной последнюю пару.) Тип `jsonb`, напротив, не сохраняет пробелы, порядок ключей и значения с дублирующимися ключами. Если во входных данных оказываются дублирующиеся ключи, сохраняется только последнее значение.

Для большинства приложений предпочтительнее хранить данные JSON в типе `jsonb` (если нет особых противопоказаний, например важны прежние предположения о порядке ключей объектов).

Postgres Pro позволяет использовать только одну кодировку символов в базе данных, поэтому данные JSON не будут полностью соответствовать спецификации, если кодировка базы данных не UTF-8. При этом нельзя будет вставить символы, непредставимые в кодировке сервера, и наоборот, допустимыми будут символы, представимые в кодировке сервера, но не в UTF-8.

RFC 7159 разрешает включать в строки JSON спецпоследовательности Unicode в виде `\u`*`XXXX`*. В функцию ввода для типа `json` эти спецпоследовательности допускаются вне зависимости от кодировки базы данных, и проверяется только правильность их синтаксиса (за `\u` должны следовать четыре шестнадцатеричных цифры). Однако функция ввода для типа `jsonb` более строгая: она не допускает спецпоследовательности Unicode для не ASCII-символов (символов после `U+007F`), если кодировка базы данных не UTF8. Тип `jsonb` также не принимает `\u0000` (так как это значение не может быть представлено в типе `text` Postgres Pro) и требует, чтобы суррогатные пары Unicode использовались для представления символов вне основной многоязыковой плоскости (BMP) правильно. Корректные спецпоследовательности Unicode преобразуются для хранения в соответствующий символ ASCII или UTF8 (это подразумевает сворачивание суррогатных пар в один символ).

> **Примечание**
>
> Многие из функций обработки JSON, описанные в Разделе 9.15, преобразуют спецпоследовательности Unicode в обычные символы, поэтому могут выдавать подобные ошибки, даже если им на вход поступает тип `json`, а не `jsonb`. То, что функция ввода в тип `json` не производит этих проверок, можно считать историческим артефактом, хотя это и позволяет просто сохранять (но не обрабатывать) в JSON спецкоды Unicode в базе данных

с кодировкой не UTF8. Вообще же, по возможности следует избегать смешения спецкодов Unicode в JSON с кодировкой базой данных не UTF8.

При преобразовании вводимого текста JSON в тип `jsonb`, примитивные типы, описанные в RFC 7159, по сути отображаются в собственные типы Postgres Pro как показано в Таблице 8.23. Таким образом, к содержимому типа `jsonb` предъявляются некоторые дополнительные требования, продиктованные ограничениями представления нижележащего типа данных, которые не распространяются ни на тип `json`, ни на формат JSON вообще. В частности, тип `jsonb` не принимает числа, выходящие за диапазон типа данных Postgres Pro `numeric`, тогда как с `json` такого ограничения нет. Такие ограничения, накладываемые реализацией, допускаются согласно RFC 7159. Однако на практике такие проблемы более вероятны в других реализациях, так как обычно примитивный тип JSON `number` представляется в виде числа с плавающей точкой двойной точности IEEE 754 (что RFC 7159 явно признаёт и допускает). При использовании JSON в качестве формата обмена данными с такими системами следует учитывать риски потери точности чисел, хранившихся в Postgres Pro.

И напротив, как показано в таблице, есть некоторые ограничения в формате ввода примитивных типов JSON, не актуальные для соответствующих типов Postgres Pro.

Таблица 8.23. Примитивные типы JSON и соответствующие им типы Postgres Pro

Примитивный тип JSON	Тип Postgres Pro	Замечания
`string`	`text`	`\u0000` не допускается, как не ASCII символ, если кодировка базы данных не UTF8
`number`	`numeric`	Значения `NaN` и `infinity` не допускаются
`boolean`	`boolean`	Допускаются только варианты `true` и `false` (в нижнем регистре)
`null`	(нет)	`NULL` в SQL имеет другой смысл

8.14.1. Синтаксис вводимых и выводимых значений JSON

Синтаксис ввода/вывода типов данных JSON соответствует стандарту RFC 7159.

Примеры допустимых выражений с типом `json` (или `jsonb`):

```
-- Простое скалярное/примитивное значение
-- Простыми значениями могут быть числа, строки в кавычках, true, false или null
SELECT '5'::json;

-- Массив из нуля и более элементов (элементы могут быть разных типов)
SELECT '[1, 2, "foo", null]'::json;

-- Объект, содержащий пары ключей и значений
-- Заметьте, что ключи объектов — это всегда строки в кавычках
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;

-- Массивы и объекты могут вкладываться произвольным образом
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

Как было сказано ранее, когда значение JSON вводится и затем выводится без дополнительной обработки, тип `json` выводит тот же текст, что поступил на вход, а `jsonb` не сохраняет семантически незначащие детали, такие как пробелы. Например, посмотрите на эти различия:

```
SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::json;
                      json
```

```
-------------------------------------------------
 {"bar": "baz", "balance": 7.77, "active":false}
(1 row)

SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::jsonb;
                       jsonb
-------------------------------------------------
 {"bar": "baz", "active": false, "balance": 7.77}
(1 row)
```

Первая семантически незначимая деталь, заслуживающая внимания: с `jsonb` числа выводятся по правилам нижележащего типа `numeric`. На практике это означает, что числа, заданные в записи с `E`, будут выведены без неё, например:

```
SELECT '{"reading": 1.230e-5}'::json, '{"reading": 1.230e-5}'::jsonb;
         json          |          jsonb
-----------------------+-------------------------
 {"reading": 1.230e-5} | {"reading": 0.00001230}
(1 row)
```

Однако как видно из этого примера, `jsonb` сохраняет конечные нули дробного числа, хотя они и не имеют семантической значимости, в частности для проверки на равенство.

8.14.2. Эффективная организация документов JSON

Представлять данные в JSON можно гораздо более гибко, чем в традиционной реляционной модели данных, что очень привлекательно там, где нет жёстких условий. И оба этих подхода вполне могут сосуществовать и дополнять друг друга в одном приложении. Однако даже для приложений, которым нужна максимальная гибкость, рекомендуется, чтобы документы JSON имели некоторую фиксированную структуру. Эта структура обычно не навязывается жёстко (хотя можно декларативно диктовать некоторые бизнес-правила), но когда она предсказуема, становится гораздо проще писать запросы, которые извлекают полезные данные из набора «документов» (информации) в таблице.

Данные JSON, как и данные любых других типов, хранящиеся в таблицах, находятся под контролем механизма параллельного доступа. Хотя хранить большие документы вполне возможно, не забывайте, что при любом изменении устанавливается блокировка всей строки (на уровне строки). Поэтому для оптимизации блокировок транзакций, изменяющих данные, стоит ограничить размер документов JSON разумными пределами. В идеале каждый документ JSON должен собой представлять атомарный информационный блок, который, согласно бизнес-логике, нельзя разделить на меньшие, индивидуально изменяемые блоки.

8.14.3. Проверки на вхождение и существование `jsonb`

Проверка *вхождения* — важная особенность типа `jsonb`, не имеющая аналога для типа `json`. Эта проверка определяет, входит ли один документ `jsonb` в другой. В следующих примерах возвращается истинное значение (кроме упомянутых исключений):

```
-- Простые скалярные/примитивные значения включают только одно идентичное значение:
SELECT '"foo"'::jsonb @> '"foo"'::jsonb;

-- Массив с правой стороны входит в массив слева:
SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;

-- Порядок элементов в массиве не важен, поэтому это условие тоже выполняется:
SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb;

-- А повторяющиеся элементы массива не имеют значения:
SELECT '[1, 2, 3]'::jsonb @> '[1, 2, 2]'::jsonb;
```

```
-- Объект с одной парой справа входит в объект слева:
SELECT '{"product": "PostgreSQL", "version": 9.4, "jsonb": true}'::jsonb @>
 '{"version": 9.4}'::jsonb;

-- Массив справа не считается входящим в
-- массив слева, хотя в последний и вложен подобный массив:
SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb;  -- выдаёт false

-- Но если добавить уровень вложенности, проверка на вхождение выполняется:
SELECT '[1, 2, [1, 3]]'::jsonb @> '[[1, 3]]'::jsonb;

-- Аналогично, это вхождением не считается:
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb;  -- выдаёт false

-- Ключ с пустым объектом на верхнем уровне входит в объект с таким ключом:
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"foo": {}}'::jsonb;
```

Общий принцип этой проверки в том, что входящий объект должен соответствовать объекту, содержащему его, по структуре и данным, возможно, после исключения из содержащего объекта лишних элементов массива или пар ключ/значение. Но помните, что порядок элементов массива для проверки на вхождение не имеет значения, а повторяющиеся элементы массива считаются только один раз.

В качестве особого исключения для требования идентичности структур, массив может содержать примитивное значение:

```
-- В этот массив входит примитивное строковое значение:
SELECT '["foo", "bar"]'::jsonb @> '"bar"'::jsonb;

-- Это исключение действует только в одну сторону -- здесь вхождения нет:
SELECT '"bar"'::jsonb @> '["bar"]'::jsonb;  -- выдаёт false
```

Для типа `jsonb` введён также оператор *существования*, который является вариацией на тему вхождения: он проверяет, является ли строка (заданная в виде значения `text`) ключом объекта или элементом массива на верхнем уровне значения `jsonb`. В следующих примерах возвращается истинное значение (кроме упомянутых исключений):

```
-- Строка существует в качестве элемента массива:
SELECT '["foo", "bar", "baz"]'::jsonb ? 'bar';

-- Строка существует в качестве ключа объекта:
SELECT '{"foo": "bar"}'::jsonb ? 'foo';

-- Значения объектов не рассматриваются:
SELECT '{"foo": "bar"}'::jsonb ? 'bar';  -- выдаёт false

-- Как и вхождение, существование определяется на верхнем уровне:
SELECT '{"foo": {"bar": "baz"}}'::jsonb ? 'bar'; -- выдаёт false

-- Строка считается существующей, если она соответствует примитивной строке JSON:
SELECT '"foo"'::jsonb ? 'foo';
```

Объекты JSON для проверок на существование и вхождение со множеством ключей или элементов подходят больше, чем массивы, так как, в отличие от массивов, они внутри оптимизируются для поиска, и поиск элемента не будет линейным.

Подсказка

Так как вхождение в JSON проверяется с учётом вложенности, правильно написанный запрос может заменить явную выборку внутренних объектов. Например, предположим, что у нас

есть столбец `doc`, содержащий объекты на верхнем уровне, и большинство этих объектов содержит поля `tags` с массивами вложенных объектов. Данный запрос найдёт записи, в которых вложенные объекты содержат ключи `"term":"paris"` и `"term":"food"`, и при этом пропустит такие ключи, находящиеся вне массива `tags`:

```
SELECT doc->'site_name' FROM websites
  WHERE doc @> '{"tags":[{"term":"paris"}, {"term":"food"}]}';
```

Этого же результата можно добиться, например, так:

```
SELECT doc->'site_name' FROM websites
  WHERE doc->'tags' @> '[{"term":"paris"}, {"term":"food"}]';
```

Но данный подход менее гибкий и часто также менее эффективный.

С другой стороны, оператор существования JSON не учитывает вложенность: он будет искать заданный ключ или элемент массива только на верхнем уровне значения JSON.

Различные операторы вхождения и существования, а также все другие операторы и функции для работы с JSON документированы в Разделе 9.15.

8.14.4. Индексация `jsonb`

Для эффективного поиска ключей или пар ключ/значение в большом количестве документов `jsonb` можно успешно применять индексы GIN. Для этого предоставляются два «класса операторов» GIN, предлагающие выбор между производительностью и гибкостью.

Класс операторов GIN по умолчанию для `jsonb` поддерживает запросы с операторами существования ключа на верхнем уровне (`?`, `?&` и `?|`) и оператором существования пути/значения (`@>`). (Подробнее семантика, реализуемая этими операторами, описана в Таблице 9.44.) Пример создания индекса с этим классом операторов:

```
CREATE INDEX idxgin ON api USING GIN (jdoc);
```

Дополнительный класс операторов GIN `jsonb_path_ops` поддерживает индексацию только для оператора `@>`. Пример создания индекса с этим классом операторов:

```
CREATE INDEX idxginp ON api USING GIN (jdoc jsonb_path_ops);
```

Рассмотрим пример таблицы, в которой хранятся документы JSON, получаемые от сторонней веб-службы, с документированным определением схемы. Типичный документ:

```
{
    "guid": "9c36adc1-7fb5-4d5b-83b4-90356a46061a",
    "name": "Angela Barton",
    "is_active": true,
    "company": "Magnafone",
    "address": "178 Howard Place, Gulf, Washington, 702",
    "registered": "2009-11-07T08:53:22 +08:00",
    "latitude": 19.793713,
    "longitude": 86.513373,
    "tags": [
        "enim",
        "aliquip",
        "qui"
    ]
}
```

Мы сохраняем эти документы в таблице `api`, в столбце `jdoc` типа `jsonb`. Если по этому столбцу создаётся GIN-индекс, он может применяться в подобных запросах:

```
-- Найти документы, в которых ключ "company" имеет значение "Magnafone"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"company": "Magnafone"}';
```

Однако в следующих запросах он не будет использоваться, потому что, несмотря на то, что оператор `?` — индексируемый, он применяется не к индексированному столбцу `jdoc` непосредственно:

```
-- Найти документы, в которых ключ "tags" содержит ключ или элемент массива "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc -> 'tags' ? 'qui';
```

И всё же, правильно применяя индексы выражений, в этом запросе можно задействовать индекс. Если запрос определённых элементов в ключе `"tags"` выполняется часто, вероятно стоит определить такой индекс:

```
CREATE INDEX idxgintags ON api USING GIN ((jdoc -> 'tags'));
```

Теперь предложение `WHERE jdoc -> 'tags' ? 'qui'` будет выполняться как применение индексируемого оператора `?` к индексируемому выражению `jdoc -> 'tags'`. (Подробнее об индексах выражений можно узнать в Разделе 11.7.)

Ещё один подход к использованию проверок на существование:

```
-- Найти документы, в которых ключ "tags" содержит элемент массива "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"tags": ["qui"]}';
```

Этот запрос может задействовать простой GIN-индекс по столбцу `jdoc`. Но заметьте, что такой индекс будет хранить копии всех ключей и значений в поле `jdoc`, тогда как индекс выражения из предыдущего примера хранит только данные внутри объекта с ключом `tags`. Хотя подход с простым индексом гораздо более гибкий (так как он поддерживает запросы по любому ключу), индексы конкретных выражений скорее всего будут меньше и быстрее, чем простые индексы.

Класс операторов `jsonb_path_ops` поддерживает только запросы с оператором `@>`, но зато он значительно производительнее класса по умолчанию `jsonb_ops`. Индекс `jsonb_path_ops` обычно гораздо меньше индекса `jsonb_ops` для тех же данных и более точен при поиске, особенно если запросы обращаются к ключам, часто встречающимся в данных. Таким образом, с ним операции поиска выполняются гораздо лучше, чем с классом операторов по умолчанию.

Техническое различие между GIN-индексами `jsonb_ops` и `jsonb_path_ops` состоит в том, что для первых создаются независимые элементы индекса для каждого ключа/значения в данных, тогда как для вторых создаются элементы только для значений. [1] По сути, каждый элемент индекса `jsonb_path_ops` представляет собой хеш значения и ключа(ей), приводящего к нему; например, при индексации `{"foo": {"bar": "baz"}}` будет создан один элемент индекса с хешем, рассчитанным по всем трём значениям: `foo`, `bar` и `baz`. Таким образом, проверка на вхождение этой структуры будет использовать крайне точный поиск по индексу, но определить, является ли `foo` ключом, с помощью такого индекса нельзя. С другой стороны, индекс `jsonb_ops` создаст три отдельных элемента индекса, представляющих `foo`, `bar` и `baz` по отдельности; для выполнения проверки на вхождение будут проверены строки таблицы, содержащие все эти три значения. Хотя GIN-индексы позволяют вычислить AND довольно эффективно, такой поиск всё же будет менее точным и более медленным, чем равнозначный поиск с `jsonb_path_ops`, особенно если любое одно из этих трёх значений содержится в большом количестве строк.

Недостаток класса `jsonb_path_ops` заключается в том, что он не учитывает в индексе структуры JSON, не содержащие никаких значений `{"a": {}}`. Для поиска по документам, содержащих такие структуры, потребуется выполнить полное сканирование индекса, что довольно долго, поэтому `jsonb_path_ops` не очень подходит для приложений, часто выполняющих такие запросы.

Тип `jsonb` также поддерживает индексы `btree` и `hash`. Они полезны, только если требуется проверять равенство JSON-документов в целом. Порядок сортировки `btree` для типа `jsonb` редко имеет большое значение, но для полноты он приводится ниже:

Объект > Массив > Логическое значение > Число > Строка > Null

Объект с n парами > Объект с n − 1 парами

[1] Поэтому понятие «значение» включает и элементы массивов, хотя в терминологии JSON иногда элементы массивов считаются отличными от значений внутри объектов.

Массив с n элементами > Массив с n - 1 элементами

Объекты с равным количеством пар сравниваются в таком порядке:

ключ-1, *значение-1*, *ключ-2* ...

Заметьте, что ключи объектов сравниваются согласно порядку при хранении; в частности, из-за того, что короткие ключи хранятся перед длинными, результаты могут оказаться несколько не интуитивными:

```
{ "aa": 1, "c": 1} > {"b": 1, "d": 1}
```

Массивы с равным числом элементом упорядочиваются аналогично:

элемент-1, *элемент-2* ...

Примитивные значения JSON сравниваются по тем же правилам сравнения, что и нижележащие типы данных Postgres Pro. Строки сравниваются с учётом порядка сортировки по умолчанию в текущей базе данных.

8.15. Массивы

Postgres Pro позволяет определять столбцы таблицы как многомерные массивы переменной длины. Элементами массивов могут быть любые встроенные или определённые пользователями типы, перечисления или составные типы. Массивы доменов в данный момент не поддерживаются.

8.15.1. Объявления типов массивов

Чтобы проиллюстрировать использование массивов, мы создадим такую таблицу:

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  integer[],
    schedule        text[][]
);
```

Как показано, для объявления типа массива к названию типа элементов добавляются квадратные скобки (`[]`). Показанная выше команда создаст таблицу `sal_emp` со столбцами типов `text` (`name`), одномерный массив с элементами `integer` (`pay_by_quarter`), представляющий квартальную зарплату работников, и двухмерный массив с элементами `text` (`schedule`), представляющий недельный график работника.

Команда `CREATE TABLE` позволяет также указать точный размер массивов, например так:

```
CREATE TABLE tictactoe (
    squares   integer[3][3]
);
```

Однако текущая реализация игнорирует все указанные размеры, т. е. фактически размер массива остаётся неопределённым.

Текущая реализация также не ограничивает число размерностей. Все элементы массивов считаются одного типа, вне зависимости от его размера и числа размерностей. Поэтому явно указывать число элементов или размерностей в команде `CREATE TABLE` имеет смысл только для документирования, на механизм работы с массивом это не влияет.

Для объявления одномерных массивов можно применять альтернативную запись с ключевым словом `ARRAY`, соответствующую стандарту SQL. Столбец `pay_by_quarter` можно было бы определить так:

```
    pay_by_quarter  integer ARRAY[4],
```

Или без указания размера массива:

```
    pay_by_quarter  integer ARRAY,
```

Заметьте, что и в этом случае Postgres Pro не накладывает ограничения на фактический размер массива.

8.15.2. Ввод значения массива

Чтобы записать значение массива в виде буквальной константы, заключите значения элементов в фигурные скобки и разделите их запятыми. (Если вам знаком C, вы найдёте, что это похоже на синтаксис инициализации структур в C.) Вы можете заключить значение любого элемента в двойные кавычки, а если он содержит запятые или фигурные скобки, это обязательно нужно сделать. (Подробнее это описано ниже.) Таким образом, общий формат константы массива выглядит так:

```
'{ значение1 разделитель значение2 разделитель ... }'
```

где *разделитель* — символ, указанный в качестве разделителя в соответствующей записи в таблице `pg_type`. Для стандартных типов данных, существующих в дистрибутиве Postgres Pro, разделителем является запятая (`,`), за исключением лишь типа `box`, в котором разделитель — точка с запятой (`;`). Каждое *значение* здесь — это либо константа типа элемента массива, либо вложенный массив. Например, константа массива может быть такой:

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

Эта константа определяет двухмерный массив 3x3, состоящий из трёх вложенных массивов целых чисел.

Чтобы присвоить элементу массива значение NULL, достаточно просто написать `NULL` (регистр символов при этом не имеет значения). Если же требуется добавить в массив строку, содержащую «NULL», это слово нужно заключить в двойные кавычки.

(Такого рода константы массивов на самом деле представляют собой всего лишь частный случай констант, описанных в Подразделе 4.1.2.7. Константа изначально воспринимается как строка и передаётся процедуре преобразования вводимого массива. При этом может потребоваться явно указать целевой тип.)

Теперь мы можем показать несколько операторов `INSERT`:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"training", "presentation"}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"breakfast", "consulting"}, {"meeting", "lunch"}}');
```

Результат двух предыдущих команд:

```
SELECT * FROM sal_emp;
name |     pay_by_quarter      |                 schedule
-----+-------------------------+-----------------------------------------
Bill |{10000,10000,10000,10000}|{{meeting,lunch},{training,presentation}}
Carol|{20000,25000,25000,25000}|{{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

В многомерных массивов число элементов в каждой размерности должно быть одинаковым; в противном случае возникает ошибка. Например:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"meeting"}}');
```

```
ОШИБКА:  для многомерных массивов должны задаваться выражения
  с соответствующими размерностями
```

Также можно использовать синтаксис конструктора `ARRAY`:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    ARRAY[10000, 10000, 10000, 10000],
    ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);

INSERT INTO sal_emp
    VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);
```

Заметьте, что элементы массива здесь — это простые SQL-константы или выражения; и поэтому, например строки будут заключаться в одинарные апострофы, а не в двойные, как в буквальной константе массива. Более подробно конструктор `ARRAY` обсуждается в Подразделе 4.2.12.

8.15.3. Обращение к массивам

Добавив данные в таблицу, мы можем перейти к выборкам. Сначала мы покажем, как получить один элемент массива. Этот запрос получает имена сотрудников, зарплата которых изменилась во втором квартале:

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];

 name
-------
 Carol
(1 row)
```

Индексы элементов массива записываются в квадратных скобках. По умолчанию в Postgres Pro действует соглашение о нумерации элементов массива с 1, то есть в массиве из *n* элементов первым считается `array[1]`, а последним — `array[`*n*`]`.

Этот запрос выдаёт зарплату всех сотрудников в третьем квартале:

```
SELECT pay_by_quarter[3] FROM sal_emp;

 pay_by_quarter
----------------
          10000
          25000
(2 rows)
```

Мы также можем получать обычные прямоугольные срезы массива, то есть подмассивы. Срез массива обозначается как *`нижняя-граница`*`:`*`верхняя-граница`* для одной или нескольких размерностей. Например, этот запрос получает первые пункты в графике Билла в первые два дня недели:

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{meeting},{training}}
(1 row)
```

Если одна из размерностей записана в виде среза, то есть содержит двоеточие, тогда срез распространяется на все размерности. Если при этом для размерности указывается только одно число (без двоеточия), в срез войдут элемент от 1 до заданного номера. Например, в этом примере `[2]` будет равнозначно `[1:2]`:

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Bill';

                schedule
-----------------------------------------
 {{meeting,lunch},{training,presentation}}
(1 row)
```

Во избежание путаницы с обращением к одному элементу, срезы лучше всегда записывать явно для всех измерений, например `[1:2][1:1]` вместо `[2][1:1]`.

Значения *`нижняя-граница`* и/или *`верхняя-граница`* в указании среза можно опустить; опущенная граница заменяется нижним или верхним пределом индексов массива. Например:

```
SELECT schedule[:2][2:] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{lunch},{presentation}}
(1 row)

SELECT schedule[:][1:1] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{meeting},{training}}
(1 row)
```

Выражение обращения к элементу массива возвратит NULL, если сам массив или одно из выражений индексов элемента равны NULL. Значение NULL также возвращается, если индекс выходит за границы массива (это не считается ошибкой). Например, если `schedule` в настоящее время имеет размерности `[1:3][1:2]`, результатом обращения к `schedule[3][3]` будет NULL. Подобным образом, при обращении к элементу массива с неправильным числом индексов возвращается NULL, а не ошибка.

Аналогично, NULL возвращается при обращении к срезу массива, если сам массив или одно из выражений, определяющих индексы элементов, равны NULL. Однако в других случаях, например, когда границы среза выходят за рамки массива, возвращается не NULL, а пустой массив (с размерностью 0). (Так сложилось исторически, что в этом срезы отличаются от обращений к обычным элементам.) Если запрошенный срез пересекает границы массива, тогда возвращается не NULL, а срез, сокращённый до области пересечения.

Текущие размеры значения массива можно получить с помощью функции `array_dims`:

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';

 array_dims
------------
 [1:2][1:2]
(1 row)
```

`array_dims` выдаёт результат типа `text`, что удобно скорее для людей, чем для программ. Размеры массива также можно получить с помощью функций `array_upper` и `array_lower`, которые возвращают соответственно верхнюю и нижнюю границу для указанной размерности:

```
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_upper
-------------
           2
(1 row)
```

`array_length` возвращает число элементов в указанной размерности массива:

```
SELECT array_length(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_length
--------------
            2
(1 row)
```

`cardinality` возвращает общее число элементов массива по всем измерениям. Фактически это число строк, которое вернёт функция `unnest`:

```
SELECT cardinality(schedule) FROM sal_emp WHERE name = 'Carol';

 cardinality
-------------
           4
(1 row)
```

8.15.4. Изменение массивов

Значение массива можно заменить полностью так:

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
    WHERE name = 'Carol';
```

или используя синтаксис `ARRAY`:

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
    WHERE name = 'Carol';
```

Также можно изменить один элемент массива:

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
    WHERE name = 'Bill';
```

или срез:

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
    WHERE name = 'Carol';
```

При этом в указании среза может быть опущена *`нижняя-граница`* и/или *`верхняя-граница`*, но только для массива, отличного от NULL, и имеющего ненулевую размерность (иначе неизвестно, какие граничные значения должны подставляться вместо опущенных).

Сохранённый массив можно расширить, определив значения ранее отсутствовавших в нём элементов. При этом все элементы, располагающиеся между существовавшими ранее и новыми, принимают значения NULL. Например, если массив `myarray` содержит 4 элемента, после присваивания значения элементу `myarray[6]` его длина будет равна 6, а `myarray[5]` будет содержать NULL. В настоящее время подобное расширение поддерживается только для одномерных, но не многомерных массивов.

Определяя элементы по индексам, можно создавать массивы, в которых нумерация элементов может начинаться не с 1. Например, можно присвоить значение выражению `myarray[-2:7]` и таким образом создать массив, в котором будут элементы с индексами от -2 до 7.

Значения массива также можно сконструировать с помощью оператора конкатенации, `||`:

```
SELECT ARRAY[1,2] || ARRAY[3,4];
 ?column?
-----------
 {1,2,3,4}
(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
```

```
      ?column?
--------------------
 {{5,6},{1,2},{3,4}}
(1 row)
```

Оператор конкатенации позволяет вставить один элемент в начало или в конец одномерного массива. Он также может принять два *N*-мерных массива или массивы размерностей *N* и *N+1*.

Когда в начало или конец одномерного массива вставляется один элемент, в образованном в результате массиве будет та же нижняя граница, что и в массиве-операнде. Например:

```
SELECT array_dims(1 || '[0:1]={2,3}'::int[]);
 array_dims
------------
 [0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);
 array_dims
------------
 [1:3]
(1 row)
```

Когда складываются два массива одинаковых размерностей, в результате сохраняется нижняя граница внешней размерности левого операнда. Выходной массив включает все элементы левого операнда, после которых добавляются все элементы правого. Например:

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
 array_dims
------------
 [1:5]
(1 row)

SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
 array_dims
------------
 [1:5][1:2]
(1 row)
```

Когда к массиву размерности *N+1* спереди или сзади добавляется *N*-мерный массив, он вставляется аналогично тому, как в массив вставляется элемент (это было описано выше). Любой *N*-мерный массив по сути является элементом во внешней размерности массива, имеющего размерность *N+1*. Например:

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
 array_dims
------------
 [1:3][1:2]
(1 row)
```

Массив также можно сконструировать с помощью функций `array_prepend`, `array_append` и `array_cat`. Первые две функции поддерживают только одномерные массивы, а `array_cat` поддерживает и многомерные. Несколько примеров:

```
SELECT array_prepend(1, ARRAY[2,3]);
 array_prepend
---------------
 {1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3);
```

```
 array_append
--------------
 {1,2,3}
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
 array_cat
-----------
 {1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
      array_cat
---------------------
 {{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
      array_cat
---------------------
 {{5,6},{1,2},{3,4}}
```

В простых случаях описанный выше оператор конкатенации предпочтительнее непосредственного вызова этих функций. Однако так как оператор конкатенации перегружен для решения всех трёх задач, возможны ситуации, когда лучше применить одну из этих функций во избежание неоднозначности. Например, рассмотрите:

```
SELECT ARRAY[1, 2] || '{3, 4}';  -- нетипизированная строка воспринимается как массив
 ?column?
-----------
 {1,2,3,4}

SELECT ARRAY[1, 2] || '7';                -- как и эта
ERROR:  malformed array literal: "7"

SELECT ARRAY[1, 2] || NULL;               -- как и буквальный NULL
 ?column?
----------
 {1,2}
(1 row)

SELECT array_append(ARRAY[1, 2], NULL);   -- это могло иметься в виду на самом деле
 array_append
--------------
 {1,2,NULL}
```

В показанных примерах анализатор запроса видит целочисленный массив с одной стороны оператора конкатенации и константу неопределённого типа с другой. Согласно своим правилам разрешения типа констант, он полагает, что она имеет тот же тип, что и другой операнд — в данном случае целочисленный массив. Поэтому предполагается, что оператор конкатенации здесь представляет функцию `array_cat`, а не `array_append`. Если это решение оказывается неверным, его можно скорректировать, приведя константу к типу элемента массива; однако может быть лучше явно использовать функцию `array_append`.

8.15.5. Поиск значений в массивах

Чтобы найти значение в массиве, необходимо проверить все его элементы. Это можно сделать вручную, если вы знаете размер массива. Например:

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
```

```
                                    pay_by_quarter[2] = 10000 OR
                                    pay_by_quarter[3] = 10000 OR
                                    pay_by_quarter[4] = 10000;
```

Однако с большим массивами этот метод становится утомительным, и к тому же он не работает, когда размер массива неизвестен. Альтернативный подход описан в Разделе 9.23. Показанный выше запрос можно было переписать так:

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

А так можно найти в таблице строки, в которых массивы содержат только значения, равные 10000:

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

Кроме того, для обращения к элементам массива можно использовать функцию `generate_subscripts`. Например так:

```
SELECT * FROM
   (SELECT pay_by_quarter,
           generate_subscripts(pay_by_quarter, 1) AS s
      FROM sal_emp) AS foo
 WHERE pay_by_quarter[s] = 10000;
```

Эта функция описана в Таблице 9.59.

Также искать в массиве значения можно, используя оператор `&&`, который проверяет, перекрывается ли левый операнд с правым. Например:

```
SELECT * FROM sal_emp WHERE pay_by_quarter && ARRAY[10000];
```

Этот и другие операторы для работы с массивами описаны в Разделе 9.18. Он может быть ускорен с помощью подходящего индекса, как описано в Разделе 11.2.

Вы также можете искать определённые значения в массиве, используя функции `array_position` и `array_positions`. Первая функция возвращает позицию первого вхождения значения в массив, а вторая — массив позиций всех его вхождений. Например:

```
SELECT array_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], 'mon');
 array_positions
-----------------
 2

SELECT array_positions(ARRAY[1, 4, 3, 1, 3, 4, 2, 1], 1);
 array_positions
-----------------
 {1,4,8}
```

> **Подсказка**
>
> Массивы — это не множества; необходимость поиска определённых элементов в массиве может быть признаком неудачно сконструированной базы данных. Возможно, вместо массива лучше использовать отдельную таблицу, строки которой будут содержать данные элементов массива. Это может быть лучше и для поиска, и для работы с большим количеством элементов.

8.15.6. Синтаксис вводимых и выводимых значений массива

Внешнее текстовое представление значения массива состоит из записи элементов, интерпретируемых по правилам ввода/вывода для типа элемента массива, и оформления структуры массива. Оформление состоит из фигурных скобок (`{` и `}`), окружающих значение массива, и знаков-разделителей между его элементами. В качестве знака-разделителя обычно

используется запятая (,), но это может быть и другой символ; он определяется параметром typdelim для типа элемента массива. Для стандартных типов данных, существующих в дистрибутиве Postgres Pro, разделителем является запятая (,), за исключением лишь типа box, в котором разделитель — точка с запятой (;). В многомерном массиве у каждой размерности (ряд, плоскость, куб и т. д.) есть свой уровень фигурных скобок, а соседние значения в фигурных скобках на одном уровне должны отделяться разделителями.

Функция вывода массива заключает значение элемента в кавычки, если это пустая строка или оно содержит фигурные скобки, знаки-разделители, кавычки, обратную косую черту, пробельный символ или это текст NULL. Кавычки и обратная косая черта, включённые в такие значения, преобразуются в спецпоследовательность с обратной косой чертой. Для числовых типов данных можно рассчитывать на то, что значения никогда не будут выводиться в кавычках, но для текстовых типов следует быть готовым к тому, что выводимое значение массива может содержать кавычки.

По умолчанию нижняя граница всех размерностей массива равна одному. Чтобы представить массивы с другими нижними границами, перед содержимым массива можно указать диапазоны индексов. Такое оформление массива будет содержать квадратные скобки ([]) вокруг нижней и верхней границ каждой размерности с двоеточием (:) между ними. За таким указанием размерности следует знак равно (=). Например:

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
 FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;

 e1 | e2
----+----
  1 |  6
(1 row)
```

Процедура вывода массива включает в результат явное указание размерностей, только если нижняя граница в одной или нескольких размерностях отличается от 1.

Если в качестве значения элемента задаётся NULL (в любом регистре), этот элемент считается равным непосредственно NULL. Если же оно включает кавычки или обратную косую черту, элементу присваивается текстовая строка «NULL». Кроме того, для обратной совместимости с версиями PostgreSQL до 8.2, параметр конфигурации array_nulls можно выключить (присвоив ему off), чтобы строки NULL не воспринимались как значения NULL.

Как было показано ранее, записывая значение массива, любой его элемент можно заключить в кавычки. Это *нужно* делать, если при разборе значения массива без кавычек возможна неоднозначность. Например, в кавычки необходимо заключать элементы, содержащие фигурные скобки, запятую (или разделитель, определённый для данного типа), кавычки, обратную косую черту, а также пробельные символы в начале или конце строки. Пустые строки и строки, содержащие одно слово NULL, также нужно заключать в кавычки. Чтобы включить кавычки или обратную косую черту в значение, заключённое в кавычки, добавьте обратную косую черту перед таким символом. С другой стороны, чтобы обойтись без кавычек, таким экранированием можно защитить все символы в данных, которые могут быть восприняты как часть синтаксиса массива.

Перед открывающей и после закрывающей скобки можно добавлять пробельные символы. Пробелы также могут окружать каждую отдельную строку значения. Во всех случаях такие пробельные символы игнорируются. Однако все пробелы в строках, заключённых в кавычки, или окружённые не пробельными символами, напротив, учитываются.

Подсказка

Записывать значения массивов в командах SQL часто бывает удобнее с помощью конструктора ARRAY (см. Подраздел 4.2.12). В ARRAY отдельные значения элементов записываются так же, как если бы они не были членами массива.

8.16. Составные типы

Составной тип представляет структуру табличной строки или записи; по сути это просто список имён полей и соответствующих типов данных. Postgres Pro позволяет использовать составные типы во многом так же, как и простые типы. Например, в определении таблицы можно объявить столбец составного типа.

8.16.1. Объявление составных типов

Ниже приведены два простых примера определения составных типов:

```
CREATE TYPE complex AS (
    r       double precision,
    i       double precision
);

CREATE TYPE inventory_item AS (
    name            text,
    supplier_id     integer,
    price           numeric
);
```

Синтаксис очень похож на `CREATE TABLE`, за исключением того, что он допускает только названия полей и их типы, какие-либо ограничения (такие как `NOT NULL`) в настоящее время не поддерживаются. Заметьте, что ключевое слово `AS` здесь имеет значение; без него система будет считать, что подразумевается другой тип команды `CREATE TYPE`, и выдаст неожиданную синтаксическую ошибку.

Определив такие типы, мы можем использовать их в таблицах:

```
CREATE TABLE on_hand (
    item      inventory_item,
    count     integer
);

INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

или функциях:

```
CREATE FUNCTION price_extension(inventory_item, integer) RETURNS numeric
AS 'SELECT $1.price * $2' LANGUAGE SQL;

SELECT price_extension(item, 10) FROM on_hand;
```

Всякий раз, когда создаётся таблица, вместе с ней автоматически создаётся составной тип, представляющий тип строки таблицы, именем которого будет имя таблицы. Например, при выполнении команды:

```
CREATE TABLE inventory_item (
    name            text,
    supplier_id     integer REFERENCES suppliers,
    price           numeric CHECK (price > 0)
);
```

будет создан составной тип `inventory_item`, в точности соответствующий тому, что был показан выше, и использовать его можно так же. Заметьте, что в текущей реализации есть один недостаток: так как с составным типом не могут быть связаны ограничения, описанные в определении таблицы ограничения *не применяются* к значениям составного типа вне таблицы. (В некоторой степени это можно обойти, используя в составных типах домены.)

8.16.2. Конструирование составных значений

Чтобы записать значение составного типа в виде текстовой константы, его поля нужно заключить в круглые скобки и разделить их запятыми. Значение любого поля можно заключить в кавычки, а если оно содержит запятые или скобки, это делать обязательно. (Подробнее об этом говорится ниже.) Таким образом, в общем виде константа составного типа записывается так:

```
'( значение1 , значение2 , ... )'
```

Например, эта запись:

```
'("fuzzy dice",42,1.99)'
```

будет допустимой для описанного выше типа `inventory_item`. Чтобы присвоить NULL одному из полей, в соответствующем месте в списке нужно оставить пустое место. Например, эта константа задаёт значение NULL для третьего поля:

```
'("fuzzy dice",42,)'
```

Если же вместо NULL требуется вставить пустую строку, нужно записать пару кавычек:

```
'("",42,)'
```

Здесь в первом поле окажется пустая строка, а в третьем — NULL.

(Такого рода константы массивов на самом деле представляют собой всего лишь частный случай констант, описанных в Подразделе 4.1.2.7. Константа изначально воспринимается как строка и передаётся процедуре преобразования составного типа. При этом может потребоваться явно указать тип, к которому будет приведена константа.)

Значения составных типов также можно конструировать, используя синтаксис выражения `ROW`. В большинстве случаев это значительно проще, чем записывать значения в строке, так как при этом не нужно беспокоиться о вложенности кавычек. Мы уже обсуждали этот метод ранее:

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

Ключевое слово ROW на самом деле может быть необязательным, если в выражении определяются несколько полей, так что эту запись можно упростить до:

```
('fuzzy dice', 42, 1.99)
('', 42, NULL)
```

Синтаксис выражения `ROW` более подробно рассматривается в Подразделе 4.2.13.

8.16.3. Обращение к составным типам

Чтобы обратиться к полю столбца составного типа, после имени столбца нужно добавить точку и имя поля, подобно тому, как указывается столбец после имени таблицы. На самом деле, эти обращения неотличимы, так что часто бывает необходимо использовать скобки, чтобы команда была разобрана правильно. Например, можно попытаться выбрать поле столбца из тестовой таблицы `on_hand` таким образом:

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;
```

Но это не будет работать, так как согласно правилам SQL имя `item` здесь воспринимается как имя таблицы, а не столбца в таблице `on_hand`. Поэтому этот запрос нужно переписать так:

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

либо указать также и имя таблицы (например, в запросе с многими таблицами), примерно так:

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

В результате объект в скобках будет правильно интерпретирован как ссылка на столбец `item`, из которого выбирается поле.

При выборке поля из значения составного типа также возможны подобные синтаксические казусы. Например, чтобы выбрать одно поле из результата функции, возвращающей составное значение, потребуется написать что-то подобное:

```
SELECT (my_func(...)).field FROM ...
```

Без дополнительных скобок в этом запросе произойдёт синтаксическая ошибка.

Специальное имя поля `*` означает «все поля»; подробнее об этом рассказывается в Подразделе 8.16.5.

8.16.4. Изменение составных типов

Ниже приведены примеры правильных команд добавления и изменения значений составных столбцов. Первые команды иллюстрируют добавление или изменение всего столбца:

```
INSERT INTO mytab (complex_col) VALUES((1.1,2.2));

UPDATE mytab SET complex_col = ROW(1.1,2.2) WHERE ...;
```

В первом примере опущено ключевое слово `ROW`, а во втором оно есть; присутствовать или отсутствовать оно может в обоих случаях.

Мы можем изменить также отдельное поле составного столбца:

```
UPDATE mytab SET complex_col.r = (complex_col).r + 1 WHERE ...;
```

Заметьте, что при этом не нужно (и на самом деле даже нельзя) заключать в скобки имя столбца, следующее сразу за предложением `SET`, но в ссылке на тот же столбец в выражении, находящемся по правую сторону знака равенства, скобки обязательны.

И мы также можем указать поля в качестве цели команды `INSERT`:

```
INSERT INTO mytab (complex_col.r, complex_col.i) VALUES(1.1, 2.2);
```

Если при этом мы не укажем значения для всех полей столбца, оставшиеся поля будут заполнены значениями NULL.

8.16.5. Использование составных типов в запросах

С составными типами в запросах связаны особые правила синтаксиса и поведение. Эти правила образуют полезные конструкции, но они могут быть неочевидными, если не понимать стоящую за ними логику.

В Postgres Pro ссылка на имя таблицы (или её псевдоним) в запросе по сути является ссылкой на составное значение текущей строки в этой таблице. Например, имея таблицу `inventory_item`, показанную выше, мы можем написать:

```
SELECT c FROM inventory_item c;
```

Этот запрос выдаёт один столбец с составным значением, и его результат может быть таким:

```
           c
------------------------
 ("fuzzy dice",42,1.99)
(1 row)
```

Заметьте, однако, что простые имена сопоставляются сначала с именами столбцов, и только потом с именами таблиц, так что такой результат получается только потому, что в таблицах запроса не оказалось столбца с именем `c`.

Обычную запись полного имени столбца вида *`имя_таблицы.имя_столбца`* можно понимать как применение выбора поля к составному значению текущей строки таблицы. (Из соображений эффективности на самом деле это реализовано по-другому.)

Когда мы пишем

```
SELECT c.* FROM inventory_item c;
```

то, согласно стандарту SQL, мы должны получить содержимое таблицы, развёрнутое в отдельные столбцы:

```
    name    | supplier_id | price
------------+-------------+-------
 fuzzy dice |          42 |  1.99
(1 row)
```

как с запросом

```
SELECT c.name, c.supplier_id, c.price FROM inventory_item c;
```

Postgres Pro применяет такое развёртывание для любых выражений с составными значениями, хотя как показано выше, необходимо заключить в скобки значение, к которому применяется `.*`, если только это не простое имя таблицы. Например, если `myfunc()` — функция, возвращающая составной тип со столбцами `a`, `b` и `c`, то эти два запроса выдадут одинаковый результат:

```
SELECT (myfunc(x)).* FROM some_table;
SELECT (myfunc(x)).a, (myfunc(x)).b, (myfunc(x)).c FROM some_table;
```

Подсказка

Postgres Pro осуществляет развёртывание столбцов фактически переводя первую форму во вторую. Таким образом, в данном примере `myfunc()` будет вызываться три раза для каждой строки и с тем, и с тем синтаксисом. Если это дорогостоящая функция и вы хотите избежать лишних вызовов, это осуществимо с таким запросом:

```
SELECT (m).* FROM (SELECT myfunc(x) AS m FROM some_table OFFSET 0) ss;
```

Предложение `OFFSET 0` добавлено, чтобы оптимизатор не упрощал подзапрос, в результате чего он может преобразоваться в форму с множественными вызовами `myfunc()`.

Запись *составное_значение*`.*` приводит к такому развёртыванию столбцов, когда она фигурирует на верхнем уровне выходного списка `SELECT`, в списке `RETURNING` команд `INSERT`/`UPDATE`/`DELETE`, в предложении `VALUES` или в конструкторе строки. Во всех других контекстах (включая вложенные в одну из этих конструкций), добавление `.*` к составному значению не меняет это значение, так как это воспринимается как «все столбцы» и поэтому выдаётся то же составное значение. Например, если функция `somefunc()` принимает в качестве аргумента составное значение, эти запросы равносильны:

```
SELECT somefunc(c.*) FROM inventory_item c;
SELECT somefunc(c) FROM inventory_item c;
```

В обоих случаях текущая строка таблицы `inventory_item` передаётся функции как один аргумент с составным значением. И хотя дополнение `.*` в этих случаях не играет роли, использовать его считается хорошим стилем, так как это ясно указывает на использование составного значения. В частности анализатор запроса воспримет `c` в записи `c.*` как ссылку на имя или псевдоним таблицы, а не имя столбца, что избавляет от неоднозначности; тогда как без `.*` неясно, означает ли `c` имя таблицы или имя столбца, и на самом деле при наличии столбца с именем `c` будет выбрано второе прочтение.

Эту концепцию демонстрирует и следующий пример, все запросы в котором действуют одинаково:

```
SELECT * FROM inventory_item c ORDER BY c;
SELECT * FROM inventory_item c ORDER BY c.*;
SELECT * FROM inventory_item c ORDER BY ROW(c.*);
```

Все эти предложения `ORDER BY` обращаются к составному значению строки, вследствие чего строки сортируются по правилам, описанным в Подразделе 9.23.6. Однако если в `inventory_item` содержится столбец с именем `c`, первый запрос будет отличаться от других, так как в нём выполнится сортировка только по данному столбцу. С показанными выше именами столбцов предыдущим запросам также равнозначны следующие:

```
SELECT * FROM inventory_item c ORDER BY ROW(c.name, c.supplier_id, c.price);
SELECT * FROM inventory_item c ORDER BY (c.name, c.supplier_id, c.price);
```

(В последнем случае используется конструктор строки, в котором опущено ключевое слово `ROW`.)

Другая особенность синтаксиса, связанная с составными значениями, состоит в том, что мы можем использовать *функциональную запись* для извлечения поля составного значения. Это легко можно объяснить тем, что записи *`поле`*`(`*`таблица`*`)` и *`таблица`*`.`*`поле`* взаимозаменяемы. Например, следующие запросы равнозначны:

```
SELECT c.name FROM inventory_item c WHERE c.price > 1000;
SELECT name(c) FROM inventory_item c WHERE price(c) > 1000;
```

Более того, если у нас есть функция, принимающая один аргумент составного типа, мы можем вызвать её в любой записи. Все эти запросы равносильны:

```
SELECT somefunc(c) FROM inventory_item c;
SELECT somefunc(c.*) FROM inventory_item c;
SELECT c.somefunc FROM inventory_item c;
```

Эта равнозначность записи с полем и функциональной записи позволяет использовать с составными типами функции, реализующие «вычисляемые поля». При этом приложению, использующему последний из предыдущих запросов, не нужно знать, что фактически `somefunc` — не настоящий столбец таблицы.

> **Подсказка**
>
> Учитывая такое поведение, будет неразумно давать функции, принимающей один аргумент составного типа, то же имя, что и одному из полей данного составного типа. Если возникнет неоднозначность, предпочтительнее окажется прочтение имени поля, так что такую функцию нельзя будет вызвать без дополнительных трюков. Например, для принудительного прочтения имени функции, потребуется дополнить это имя схемой, то есть, написать *`схема`*`.`*`функция`*`(`*`составное_значение`*`)`.

8.16.6. Синтаксис вводимых и выводимых значений составного типа

Внешнее текстовое представление составного значения состоит из записи элементов, интерпретируемых по правилам ввода/вывода для соответствующих типов полей, и оформления структуры составного типа. Оформление состоит из круглых скобок (`(` и `)`) окружающих всё значение, и запятых (`,`) между его элементами. Пробельные символы вне скобок игнорируются, но внутри они считаются частью соответствующего элемента и могут учитываться или не учитываться в зависимости от правил преобразования вводимых данных для типа этого элемента. Например, в записи:

```
'(  42)'
```

пробелы будут игнорироваться, если соответствующее поле имеет целочисленный тип, но не текстовый.

Как было показано ранее, записывая составное значение, любой его элемент можно заключить в кавычки. Это *нужно* делать, если при разборе этого значения без кавычек возможна неоднозначность. Например, в кавычки нужно заключать элементы, содержащие скобки, кавычки, запятую или обратную косую черту. Чтобы включить в поле составного значения, заключённое в кавычки, такие символы, как кавычки или обратная косая черта, перед ними нужно добавить обратную косую черту. (Кроме того, продублированные кавычки в значении поля, заключённого в кавычки, воспринимаются как одинарные, подобно апострофам в строках SQL.) С другой стороны, можно обойтись без кавычек, защитив все символы в данных, которые могут быть восприняты как часть синтаксиса составного значения, с помощью спецпоследовательностей.

Значение NULL в этой записи представляется пустым местом (когда между запятыми или скобками нет никаких символов). Чтобы ввести именно пустую строку, а не NULL, нужно написать "".

Функция вывода составного значения заключает значения полей в кавычки, если они представляют собой пустые строки, либо содержат скобки, запятые, кавычки или обратную косую черту, либо состоят из одних пробелов. (В последнем случае можно обойтись без кавычек, но они добавляются для удобочитаемости.) Кавычки и обратная косая черта, заключённые в значения полей, при выводе дублируются.

Примечание

Помните, что написанная SQL-команда прежде всего интерпретируется как текстовая строка, а затем как составное значение. Вследствие этого число символов обратной косой черты удваивается (если используются спецпоследовательности). Например, чтобы ввести в поле составного столбца значение типа `text` с обратной косой чертой и кавычками, команду нужно будет записать так:

```
INSERT ... VALUES ('("\"\\")');
```

Сначала обработчик спецпоследовательностей удаляет один уровень обратной косой черты, так что анализатор составного значения получает на вход `("\"\\")`. В свою очередь, он передаёт эту строку процедуре ввода значения типа `text`, где она преобразуются в `"\`. (Если бы мы работали с типом данных, процедура ввода которого также интерпретирует обратную косую черту особым образом, например `bytea`, нам могло бы понадобиться уже восемь таких символов, чтобы сохранить этот символ в поле составного значения.) Во избежание такого дублирования спецсимволов строки можно заключать в доллары (см. Подраздел 4.1.2.4).

Подсказка

Записывать составные значения в командах SQL часто бывает удобнее с помощью конструктора `ROW`. В `ROW` отдельные значения элементов записываются так же, как если бы они не были членами составного выражения.

8.17. Диапазонные типы

Диапазонные типы представляют диапазоны значений некоторого типа данных (он также называется *подтипом* диапазона). Например, диапазон типа `timestamp` может представлять временной интервал, когда зарезервирован зал заседаний. В данном случае типом данных будет `tsrange` (сокращение от «timestamp range»), а подтипом — `timestamp`. Подтип должен быть полностью упорядочиваемым, чтобы можно было однозначно определить, где находится значение по отношению к диапазону: внутри, до или после него.

Диапазонные типы полезны тем, что позволяют представить множество возможных значений в одной структуре данных и чётко выразить такие понятия, как пересечение диапазонов. Наиболее очевидный вариант их использования — применять диапазоны даты и времени для составления расписания, но также полезными могут оказаться диапазоны цен, интервалы измерений и т. д.

8.17.1. Встроенные диапазонные типы

Postgres Pro имеет следующие встроенные диапазонные типы:

- `int4range` — диапазон подтипа `integer`
- `int8range` — диапазон подтипа `bigint`
- `numrange` — диапазон подтипа `numeric`

- `tsrange` — диапазон подтипа `timestamp without time zone`
- `tstzrange` — диапазон подтипа `timestamp with time zone`
- `daterange` — диапазон подтипа `date`

Помимо этого, вы можете определять собственные типы; подробнее это описано в CREATE TYPE.

8.17.2. Примеры

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
    (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');

-- Вхождение
SELECT int4range(10, 20) @> 3;

-- Перекрытие
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- Получение верхней границы
SELECT upper(int8range(15, 25));

-- Вычисление пересечения
SELECT int4range(10, 20) * int4range(15, 25);

-- Является ли диапазон пустым?
SELECT isempty(numrange(1, 5));
```

Полный список операторов и функций, предназначенных для диапазонных типов, приведён в Таблице 9.50 и Таблице 9.51.

8.17.3. Включение и исключение границ

Любой непустой диапазон имеет две границы, верхнюю и нижнюю, и включает все точки между этими значениями. В него также может входить точка, лежащая на границе, если диапазон включает эту границу. И наоборот, если диапазон не включает границу, считается, что точка, лежащая на этой границе, в него не входит.

В текстовой записи диапазона включение нижней границы обозначается символом «`[`», а исключением — символом «`(`». Для верхней границы включение обозначается аналогично, символом «`]`», а исключение — символом «`)`». (Подробнее это описано в Подразделе 8.17.5.)

Для проверки, включается ли нижняя или верхняя граница в диапазон, предназначены функции `lower_inc` и `upper_inc`, соответственно.

8.17.4. Неограниченные (бесконечные) диапазоны

Нижнюю границу диапазона можно опустить и определить тем самым диапазон, включающий все значения, лежащие ниже верхней границы, например: `(,3]`. Подобным образом, если не определить верхнюю границу, в диапазон войдут все значения, лежащие выше нижней границы. Если же опущена и нижняя, и верхняя границы, такой диапазон будет включать все возможные значения своего подтипа. Указание отсутствующей границы как включаемой в диапазон автоматически преобразуется в исключающее; например, `[,]` преобразуется в `(,)`. Можно воспринимать отсутствующие значения как плюс/минус бесконечность, но всё же это особые значения диапазонного типа, которые охватывают и возможные для подтипа значения плюс/минус бесконечность.

Для подтипов, в которых есть понятие «бесконечность», `infinity` может использоваться в качестве явного значения границы. При этом, например, в диапазон `[today,infinity)` с подтипом

`timestamp` не будет входить специальное значение `infinity` данного подтипа, однако это значение будет входить в диапазон `[today,infinity]`, как и в диапазоны `[today,)` и `[today,]`.

Проверить, определена ли верхняя или нижняя граница, можно с помощью функций `lower_inf` и `upper_inf`, соответственно.

8.17.5. Ввод/вывод диапазонов

Вводимое значение диапазона должно записываться в одной из следующих форм:

```
(нижняя-граница,верхняя-граница)
(нижняя-граница,верхняя-граница]
[нижняя-граница,верхняя-граница)
[нижняя-граница,верхняя-граница]
empty
```

Тип скобок (квадратные или круглые) определяет, включаются ли в диапазон соответствующие границы, как описано выше. Заметьте, что последняя форма содержит только слово `empty` и определяет пустой диапазон (диапазон, не содержащий точек).

Здесь *`нижняя-граница`* может быть строкой с допустимым значением подтипа или быть пустой (тогда диапазон будет без нижней границы). Аналогично, *`верхняя-граница`* может задаваться одним из значений подтипа или быть неопределённой (пустой).

Любое значение границы диапазона можно заключить в кавычки (`"`). А если значение содержит круглые или квадратные скобки, запятые, кавычки или обратную косую черту, использовать кавычки необходимо, чтобы эти символы не рассматривались как часть синтаксиса диапазона. Чтобы включить в значение границы диапазона, заключённое в кавычки, такие символы, как кавычки или обратная косая черта, перед ними нужно добавить обратную косую черту. (Кроме того, продублированные кавычки в значении диапазона, заключённого в кавычки, воспринимаются как одинарные, подобно апострофам в строках SQL.) С другой стороны, можно обойтись без кавычек, защитив все символы в данных, которые могут быть восприняты как часть синтаксиса диапазона, с помощью спецпоследовательностей. Чтобы задать в качестве границы пустую строку, нужно ввести `""`, так как пустая строка без кавычек будет означать отсутствие границы.

Пробельные символы до и после определения диапазона игнорируются, но когда они присутствуют внутри скобок, они воспринимаются как часть значения верхней или нижней границы. (Хотя они могут также игнорироваться в зависимости от подтипа диапазона.)

> **Примечание**
>
> Эти правила очень похожи на правила записи значений для полей составных типов. Дополнительные замечания приведены в Подразделе 8.16.6.

Примеры:

```
-- в диапазон включается 3, не включается 7 и включаются все точки между ними
SELECT '[3,7)'::int4range;

-- в диапазон не включаются 3 и 7, но включаются все точки между ними
SELECT '(3,7)'::int4range;

-- в диапазон включается только одно значение 4
SELECT '[4,4]'::int4range;

-- диапазон не включает никаких точек (нормализация заменит его определение
--  на 'empty')
SELECT '[4,4)'::int4range;
```

8.17.6. Конструирование диапазонов

Для каждого диапазонного типа определена функция конструктора, имеющая то же имя, что и данный тип. Использовать этот конструктор обычно удобнее, чем записывать текстовую константу диапазона, так как это избавляет от потребности в дополнительных кавычках. Функция конструктора может принимать два или три параметра. Вариант с двумя параметрами создаёт диапазон в стандартной форме (нижняя граница включается, верхняя исключается), тогда как для варианта с тремя параметрами включение границ определяется третьим параметром. Третий параметр должен содержать одну из строк: «`()`», «`(]`», «`[)`» или «`[]`». Например:

```
-- Полная форма: нижняя граница, верхняя граница и текстовая строка, определяющая
-- включение/исключение границ.
SELECT numrange(1.0, 14.0, '(]');

-- Если третий аргумент опущен, подразумевается '[)'.
SELECT numrange(1.0, 14.0);

-- Хотя здесь указывается '(]', при выводе значение будет приведено к
-- каноническому виду, так как int8range — тип дискретного диапазона (см. ниже).
SELECT int8range(1, 14, '(]');

-- Когда вместо любой границы указывается NULL, соответствующей границы
--  у диапазона не будет.
SELECT numrange(NULL, 2.2);
```

8.17.7. Типы дискретных диапазонов

Дискретным диапазоном считается диапазон, для подтипа которого однозначно определён «шаг», как например для типов `integer` и `date`. Значения этих двух типов можно назвать соседними, когда между ними нет никаких других значений. В непрерывных диапазонах, напротив, всегда (или почти всегда) можно найти ещё одно значение между двумя данными. Например, непрерывным диапазоном будет диапазон с подтипами `numeric` и `timestamp`. (Хотя `timestamp` имеет ограниченную точность, то есть теоретически он является дискретным, но всё же лучше считать его непрерывным, так как шаг его обычно не определён.)

Можно также считать дискретным подтип диапазона, в котором чётко определены понятия «следующего» и «предыдущего» элемента для каждого значения. Такие определения позволяют преобразовывать границы диапазона из включаемых в исключаемые, выбирая следующий или предыдущий элемент вместо заданного значения. Например, диапазоны целочисленного типа `[4,8]` и `(3,9)` описывают одно и то же множество значений; но для диапазона подтипа numeric это не так.

Для типа дискретного диапазона определяется функция *канонизации*, учитывающая размер шага для данного подтипа. Задача этой функции — преобразовать равнозначные диапазоны к единственному представлению, в частности нормализовать включаемые и исключаемые границы. Если функция канонизации не определена, диапазоны с различным определением будут всегда считаться разными, даже когда они на самом деле представляют одно множество значений.

Для встроенных типов `int4range`, `int8range` и `daterange` каноническое представление включает нижнюю границу и не включает верхнюю; то есть диапазон приводится к виду `[)`. Однако для нестандартных типов можно использовать и другие соглашения.

8.17.8. Определение новых диапазонных типов

Пользователи могут определять собственные диапазонные типы. Это может быть полезно, когда нужно использовать диапазоны с подтипами, для которых нет встроенных диапазонных типов. Например, можно определить новый тип диапазона для подтипа `float8`:

```
CREATE TYPE floatrange AS RANGE (
    subtype = float8,
```

```
    subtype_diff = float8mi
);

SELECT '[1.234, 5.678]'::floatrange;
```

Так как для `float8` осмысленное значение «шага» не определено, функция канонизации в данном примере не задаётся.

Определяя собственный диапазонный тип, вы также можете выбрать другие правила сортировки или класс оператора B-дерева для его подтипа, что позволит изменить порядок значений, от которого зависит, какие значения попадают в заданный диапазон.

Если подтип можно рассматривать как дискретный, а не непрерывный, в команде `CREATE TYPE` следует также задать функцию `канонизации`. Этой функции будет передаваться значение диапазона, а она должна вернуть равнозначное значение, но, возможно, с другими границами и форматированием. Для двух диапазонов, представляющих одно множество значений, например, целочисленные диапазоны `[1, 7]` и `[1, 8)`, функция канонизации должна выдавать один результат. Какое именно представление будет считаться каноническим, не имеет значения — главное, чтобы два равнозначных диапазона, отформатированных по-разному, всегда преобразовывались в одно значение с одинаковым форматированием. Помимо исправления формата включаемых/исключаемых границ, функция канонизации может округлять значения границ, если размер шага превышает точность хранения подтипа. Например, в типе диапазона для подтипа `timestamp` можно определить размер шага, равный часу, тогда функция канонизации должна будет округлить границы, заданные, например с точностью до минут, либо вместо этого выдать ошибку.

Помимо этого, для любого диапазонного типа, ориентированного на использование с индексами GiST или SP-GiST, должна быть определена разница значений подтипов, функция `subtype_diff`. (Индекс сможет работать и без `subtype_diff`, но в большинстве случаев это будет не так эффективно.) Эта функция принимает на вход два значения подтипа и возвращает их разницу (т. е. *`X`* минус *`Y`*) в значении типа `float8`. В показанном выше примере может использоваться функция `float8mi`, определяющая нижележащую реализацию обычного оператора «минус» для типа `float8`, но для другого подтипа могут потребоваться дополнительные преобразования. Иногда для представления разницы в числовом виде требуется ещё и творческий подход. Функция `subtype_diff`, насколько это возможно, должна быть согласована с порядком сортировки, вытекающим из выбранных правил сортировки и класса оператора; то есть, её результат должен быть положительным, если согласно порядку сортировки первый её аргумент больше второго.

Ещё один, не столь тривиальный пример функции `subtype_diff`:

```
CREATE FUNCTION time_subtype_diff(x time, y time) RETURNS float8 AS
'SELECT EXTRACT(EPOCH FROM (x - y))' LANGUAGE sql STRICT IMMUTABLE;

CREATE TYPE timerange AS RANGE (
    subtype = time,
    subtype_diff = time_subtype_diff
);

SELECT '[11:10, 23:00]'::timerange;
```

Дополнительные сведения о создании диапазонных типов можно найти в описании CREATE TYPE.

8.17.9. Индексация

Для столбцов, имеющих диапазонный тип, можно создать индексы GiST и SP-GiST. Например, так создаётся индекс GiST:

```
CREATE INDEX reservation_idx ON reservation USING GIST (during);
```

Индекс GiST или SP-GiST помогает ускорить запросы со следующими операторами: `=`, `&&`, `<@`, `@>`, `<<`, `>>`, `-|-`, `&<` и `&>` (дополнительно о них можно узнать в Таблице 9.50.

Кроме того, для таких столбцов можно создать индексы на основе хеша и B-деревьев. Для индексов таких типов полезен по сути только один оператор диапазона — равно. Порядок сортировки B-дерева определяется для значений диапазона соответствующими операторами < и >, но этот порядок может быть произвольным и он не очень важен в реальном мире. Поддержка B-деревьев и хешей диапазонными типами нужна в основном для сортировки и хеширования при выполнении запросов, но не для создания самих индексов.

8.17.10. Ограничения для диапазонов

Тогда как для скалярных значений естественным ограничением является UNIQUE, оно обычно не подходит для диапазонных типов. Вместо этого чаще оказываются полезнее ограничения-исключения (см. CREATE TABLE ... CONSTRAINT ... EXCLUDE). Такие ограничения позволяют, например определить условие «непересечения» диапазонов. Например:

```
CREATE TABLE reservation (
    during tsrange,
    EXCLUDE USING GIST (during WITH &&)
);
```

Это ограничение не позволит одновременно сохранить в таблице несколько диапазонов, которые накладываются друг на друга:

```
INSERT INTO reservation VALUES
    ('[2010-01-01 11:30, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO reservation VALUES
    ('[2010-01-01 14:45, 2010-01-01 15:45)');
ОШИБКА:  конфликтующее значение ключа нарушает ограничение-исключение
 "reservation_during_excl"
ПОДРОБНОСТИ: Ключ (during)=(["2010-01-01 14:45:00","2010-01-01 15:45:00"))
конфликтует с существующим ключом (during)=(["2010-01-01 11:30:00","2010-01-01
 15:00:00"))
```

Для максимальной гибкости в ограничении-исключении можно сочетать простые скалярные типы данных с диапазонами, используя расширение btree_gist. Например, если btree_gist установлено, следующее ограничение не будет допускать пересекающиеся диапазоны, только если совпадают также и номера комнат:

```
CREATE EXTENSION btree_gist;
CREATE TABLE room_reservation (
    room text,
    during tsrange,
    EXCLUDE USING GIST (room WITH =, during WITH &&)
);

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:00, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:30, 2010-01-01 15:30)');
ОШИБКА:  конфликтующее значение ключа нарушает ограничение-исключение
  "room_reservation_room_during_excl"
ПОДРОБНОСТИ:  Ключ (room, during)=(123A, [ 2010-01-01 14:30:00,
  2010-01-01 15:30:00 )) конфликтует
с существующим ключом (room, during)=(123A, ["2010-01-01 14:00:00","2010-01-01
 15:00:00")).

INSERT INTO room_reservation VALUES
    ('123B', '[2010-01-01 14:30, 2010-01-01 15:30)');
```

```
INSERT 0 1
```

8.18. Идентификаторы объектов

Идентификатор объекта (Object Identifier, OID) используется внутри Postgres Pro в качестве первичного ключа различных системных таблиц. В пользовательские таблицы столбец OID добавляется, только если при создании таблицы указывается `WITH OIDS` или включён параметр конфигурации default_with_oids. Идентификатор объекта представляется в типе `oid`. Также для типа `oid` определены следующие псевдонимы: `regproc`, `regprocedure`, `regoper`, `regoperator`, `regclass`, `regtype`, `regrole`, `regnamespace`, `regconfig` и `regdictionary`. Обзор этих типов приведён в Таблице 8.24.

В настоящее время тип `oid` реализован как четырёхбайтное целое. Таким образом, значение этого типа может быть недостаточно большим для обеспечения уникальности в базе данных или даже в отдельных больших таблицах. Поэтому в пользовательских таблицах использовать столбец типа OID в качестве первичного ключа не рекомендуется. Лучше всего ограничить применение этого типа обращениями к системным таблицам.

Для самого типа `oid` помимо сравнения определены всего несколько операторов. Однако его можно привести к целому и затем задействовать в обычных целочисленных вычислениях. (При этом следует опасаться путаницы со знаковыми/беззнаковыми значениями.)

Типы-псевдонимы OID сами по себе не вводят новых операций и отличаются только специализированными функциями ввода/вывода. Эти функции могут принимать и выводить не просто числовые значения, как тип `oid`, а символические имена системных объектов. Эти типы позволяют упростить поиск объектов по значениям OID. Например, чтобы выбрать из `pg_attribute` строки, относящиеся к таблице `mytable`, можно написать:

```
SELECT * FROM pg_attribute WHERE attrelid = 'mytable'::regclass;
```

вместо:

```
SELECT * FROM pg_attribute
  WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'mytable');
```

Хотя второй вариант выглядит не таким уж плохим, но это лишь очень простой запрос. Если же потребуется выбрать правильный OID, когда таблица `mytable` есть в нескольких схемах, вложенный подзапрос будет гораздо сложнее. Преобразователь вводимого значения типа `regclass` находит таблицу согласно заданному пути поиска схем, так что он делает «всё правильно» автоматически. Аналогично, приведя идентификатор таблицы к типу `regclass`, можно получить символическое представление числового кода.

Таблица 8.24. Идентификаторы объектов

Имя	Ссылки	Описание	Пример значения
`oid`	any	числовой идентификатор объекта	`564182`
`regproc`	`pg_proc`	имя функции	`sum`
`regprocedure`	`pg_proc`	функция с типами аргументов	`sum(int4)`
`regoper`	`pg_operator`	имя оператора	`+`
`regoperator`	`pg_operator`	оператор с типами аргументов	`*(integer,integer)` или `-(NONE,integer)`
`regclass`	`pg_class`	имя отношения	`pg_type`
`regtype`	`pg_type`	имя типа данных	`integer`
`regrole`	`pg_authid`	имя роли	`smithee`
`regnamespace`	`pg_namespace`	пространство имён	`pg_catalog`

Имя	Ссылки	Описание	Пример значения
regconfig	pg_ts_config	конфигурация текстового поиска	english
regdictionary	pg_ts_dict	словарь текстового поиска	simple

Все типы псевдонимов OID для объектов, сгруппированных в пространство имён, принимают имена, дополненные именем схемы, и выводят имена со схемой, если данный объект нельзя будет найти в текущем пути поиска без имени схемы. Типы regproc и regoper принимают только уникальные вводимые имена (не перегруженные), что ограничивает их применимость; в большинстве случаев лучше использовать regprocedure или regoperator. Для типа regoperator в записи унарного оператора неиспользуемый операнд заменяется словом NONE.

Дополнительным свойством большинства типов псевдонимов OID является образование зависимостей. Когда в сохранённом выражении фигурирует константа одного из этих типов (например, в представлении или в значении столбца по умолчанию), это создаёт зависимость от целевого объекта. Например, если значение по умолчанию определяется выражением nextval('my_seq'::regclass), Postgres Pro понимает, что это выражение зависит от последовательности my_seq, и не позволит удалить последовательность раньше, чем будет удалено это выражение. Единственным исключением является тип regrole. Константы этого типа в таких выражениях не допускаются.

> **Примечание**
>
> Типы псевдонимов OID не полностью следуют правилам изоляции транзакций. Планировщик тоже воспринимает их как простые константы, что может привести к неоптимальному планированию запросов.

Есть ещё один тип системных идентификаторов, xid, представляющий идентификатор транзакции (сокращённо xact). Этот тип имеют системные столбцы xmin и xmax. Идентификаторы транзакций определяются 32-битными числами.

Третий тип идентификаторов, используемых в системе, — cid, идентификатор команды (command identifier). Этот тип данных имеют системные столбцы cmin и cmax. Идентификаторы команд — это тоже 32-битные числа.

И наконец, последний тип системных идентификаторов — tid, идентификатор строки/кортежа (tuple identifier). Этот тип данных имеет системный столбец ctid. Идентификатор кортежа представляет собой пару (из номера блока и индекса кортежа в блоке), идентифицирующую физическое расположение строки в таблице.

(Подробнее о системных столбцах рассказывается в Разделе 5.4.)

8.19. Тип pg_lsn

Тип данных pg_lsn может применяться для хранения значения LSN (последовательный номер в журнале, Log Sequence Number), которое представляет собой указатель на позицию в журнале WAL. Этот тип содержит XLogRecPtr и является внутренним системным типом Postgres Pro.

Технически LSN — это 64-битное целое, представляющее байтовое смещение в потоке журнала предзаписи. Он выводится в виде двух шестнадцатеричных чисел до 8 цифр каждое, через косую черту, например: 16/B374D848. Тип pg_lsn поддерживает стандартные операторы сравнения, такие как = и >. Можно также вычесть один LSN из другого с помощью оператора –; результатом будет число байт между этими двумя позициями в журнале предзаписи.

8.20. Псевдотипы

В систему типов Postgres Pro включены несколько специальных элементов, которые в совокупности называются *псевдотипами*. Псевдотип нельзя использовать в качестве типа данных столбца, но можно объявить функцию с аргументом или результатом такого типа. Каждый из существующих псевдотипов полезен в ситуациях, когда характер функции не позволяет просто получить или вернуть определённый тип данных SQL. Все существующие псевдотипы перечислены в Таблице 8.25.

Таблица 8.25. Псевдотипы

Имя	Описание
any	Указывает, что функция принимает любой вводимый тип данных.
anyelement	Указывает, что функция принимает любой тип данных (см. Подраздел 35.2.5).
anyarray	Указывает, что функция принимает любой тип массива (см. Подраздел 35.2.5).
anynonarray	Указывает, что функция принимает любой тип данных, кроме массивов (см. Подраздел 35.2.5).
anyenum	Указывает, что функция принимает любое перечисление (см. Подраздел 35.2.5 и Раздел 8.7).
anyrange	Указывает, что функция принимает любой диапазонный тип данных (см. Подраздел 35.2.5 и Раздел 8.17).
cstring	Указывает, что функция принимает или возвращает строку в стиле C.
internal	Указывает, что функция принимает или возвращает внутренний серверный тип данных.
language_handler	Обработчик процедурного языка объявляется как возвращающий тип language_handler .
fdw_handler	Обработчик обёртки сторонних данных объявляется как возвращающий тип fdw_handler .
index_am_handler	Обработчик метода доступа индекса объявляется как возвращающий тип index_am_handler .
tsm_handler	Обработчик метода выборки из таблицы объявляется как возвращающий тип tsm_handler .
record	Указывает, что функция принимает или возвращает неопределённый тип строки.
trigger	Триггерная функция объявляется как возвращающая тип trigger.
event_trigger	Функция событийного триггера объявляется как возвращающая тип event_trigger .
pg_ddl_command	Обозначает представление команд DDL, доступное событийным триггерам.
void	Указывает, что функция не возвращает значение.
unknown	Обозначает ещё не распознанный тип, например простую строковую константу.

Имя	Описание
opaque	Устаревший тип, который раньше использовался во многих вышеперечисленных случаях.

Функции, написанные на языке C (встроенные или динамически загружаемые), могут быть объявлены с параметрами или результатами любого из этих типов. Ответственность за безопасное поведение функции с аргументами таких типов ложится на разработчика функции.

Функции, написанные на процедурных языках, могут использовать псевдотипы, только если это позволяет соответствующий язык. В настоящее время большинство процедурных языков запрещают использовать псевдотипы в качестве типа аргумента и позволяют использовать для результатов только типы `void` и `record` (и `trigger` или `event_trigger`, когда функция реализует триггер или событийный триггер). Некоторые языки также поддерживают полиморфные функции с типами `anyelement`, `anyarray`, `anynonarray`, `anyenum` и `anyrange`.

Псевдотип `internal` используется в объявлениях функций, предназначенных только для внутреннего использования в СУБД, но не для прямого вызова в запросах SQL. Если у функции есть как хотя бы один аргумент типа `internal`, её нельзя будет вызывать из SQL. Чтобы сохранить типобезопасность при таком ограничении, следуйте важному правилу: не создавайте функцию, возвращающую результат типа `internal`, если у неё нет ни одного аргумента `internal`.

Глава 9. Функции и операторы

Postgres Pro предоставляет огромное количество функций и операторов для встроенных типов данных. Кроме того, пользователи могут определять свои функции и операторы, как описано в Части V. Просмотреть все существующие функции и операторы можно в psql с помощью команд `\df` и `\do`, соответственно.

Если для вас важна переносимость, учтите, что практически все функции и операторы, описанные в этой главе, за исключением простейших арифметических и операторов сравнения, а также явно отмеченных функций, не описаны в стандарте SQL. Тем не менее частично эта расширенная функциональность присутствует и в других СУБД SQL и во многих случаях различные реализации одинаковых функций оказываются аналогичными и совместимыми. В этой главе не описываются абсолютно все функции; некоторые дополнительные функции рассматриваются в других разделах документации.

9.1. Логические операторы

Набор логических операторов включает обычные:

```
AND
OR
NOT
```

В SQL работает логическая система с тремя состояниями: true (истина), false (ложь) и `NULL`, «неопределённое» состояние. Рассмотрите следующие таблицы истинности:

a	*b*	*a* **AND** *b*	*a* **OR** *b*
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	**NOT** *a*
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Операторы `AND` и `OR` коммутативны, то есть от перемены мест операндов результат не меняется. Однако значение может иметь порядок вычисления подвыражений. Подробнее это описано в Подразделе 4.2.14.

9.2. Функции и операторы сравнения

Набор операторов сравнения включает обычные операторы, перечисленные в Таблице 9.1.

Таблица 9.1. Операторы сравнения

Оператор	Описание
<	меньше
>	больше
<=	меньше или равно
>=	больше или равно
=	равно

Оператор	Описание
<> или !=	не равно

> **Примечание**
>
> Оператор != преобразуется в <> на стадии разбора запроса. Как следствие, реализовать операторы != и <> по-разному невозможно.

Операторы сравнения определены для всех типов данных, для которых они имеют смысл. Все операторы сравнения представляют собой бинарные операторы, возвращающие значения типа boolean; при этом выражения вида 1 < 2 < 3 недопустимы (так как не существует оператора <, который бы сравнивал булево значение с 3).

Существует также несколько предикатов сравнения; они приведены в Таблице 9.2. Они работают подобно операторам, но имеют особый синтаксис, установленный стандартом SQL.

Таблица 9.2. Предикаты сравнения

Предикат	Описание
a BETWEEN *x* AND *y*	между
a NOT BETWEEN *x* AND *y*	не между
a BETWEEN SYMMETRIC *x* AND *y*	между, после сортировки сравниваемых значений
a NOT BETWEEN SYMMETRIC *x* AND *y*	не между, после сортировки сравниваемых значений
a IS DISTINCT FROM *b*	не равно, при этом NULL воспринимается как обычное значение
a IS NOT DISTINCT FROM *b*	равно, при этом NULL воспринимается как обычное значение
выражение IS NULL	эквивалентно NULL
выражение IS NOT NULL	не эквивалентно NULL
выражение ISNULL	эквивалентно NULL (нестандартный синтаксис)
выражение NOTNULL	не эквивалентно NULL (нестандартный синтаксис)
логическое_выражение IS TRUE	истина
логическое_выражение IS NOT TRUE	ложь или неопределённость
логическое_выражение IS FALSE	ложь
логическое_выражение IS NOT FALSE	истина или неопределённость
логическое_выражение IS UNKNOWN	неопределённость
логическое_выражение IS NOT UNKNOWN	истина или ложь

Предикат BETWEEN упрощает проверки интервала:

a BETWEEN *x* AND *y*

равнозначно выражению

a >= *x* AND *a* <= *y*

Заметьте, что BETWEEN считает, что границы интервала также включаются в интервал. NOT BETWEEN выполняет противоположное сравнение:

a NOT BETWEEN *x* AND *y*

равнозначно выражению

a < *x* OR *a* > *y*

Предикат BETWEEN SYMMETRIC аналогичен BETWEEN, за исключением того, что аргумент слева от AND не обязательно должен быть меньше или равен аргументу справа. Если это не так, аргументы автоматически меняются местами, так что всегда подразумевается непустой интервал.

Обычные операторы сравнения выдают NULL (что означает «неопределённость»), а не true или false, когда любое из сравниваемых значений NULL. Например, 7 = NULL выдаёт NULL, так же, как и 7 <> NULL. Когда это поведение нежелательно, можно использовать предикаты IS [NOT] DISTINCT FROM:

a IS DISTINCT FROM *b*
a IS NOT DISTINCT FROM *b*

Для значений не NULL условие IS DISTINCT FROM работает так же, как оператор <>. Однако если оба сравниваемых значения NULL, результат будет false, и только если одно из значений NULL, возвращается true. Аналогично, условие IS NOT DISTINCT FROM равносильно = для значений не NULL, но возвращает true, если оба сравниваемых значения NULL, и false в противном случае. Таким образом, эти предикаты по сути работают с NULL, как с обычным значением, а не с «неопределённостью».

Для проверки, содержит ли значение NULL или нет, используются предикаты:

выражение IS NULL
выражение IS NOT NULL

или равнозначные (но нестандартные) предикаты:

выражение ISNULL
выражение NOTNULL

Заметьте, что проверка *выражение* = NULL *не* будет работать, так как NULL считается не «равным» NULL. (Значение NULL представляет неопределённость, и равны ли две неопределённости, тоже не определено.)

> **Подсказка**
>
> Некоторые приложения могут ожидать, что *выражение* = NULL вернёт true, если результатом *выражения* является NULL. Такие приложения настоятельно рекомендуется исправить и привести в соответствие со стандартом SQL. Однако в случаях, когда это невозможно, это поведение можно изменить с помощью параметра конфигурации transform_null_equals. Когда этот параметр включён, Postgres Pro преобразует условие x = NULL в x IS NULL.

Если *выражение* возвращает табличную строку, тогда IS NULL будет истинным, когда само выражение — NULL или все поля строки — NULL, а IS NOT NULL будет истинным, когда само выражение не NULL, и все поля строки так же не NULL. Вследствие такого определения, IS NULL и IS NOT NULL не всегда будут возвращать взаимодополняющие результаты для таких выражений; в частности такие выражения со строками, одни поля которых NULL, а другие не NULL, будут ложными одновременно. В некоторых случаях имеет смысл написать *строка* IS DISTINCT FROM NULL или *строка* IS NOT DISTINCT FROM NULL, чтобы просто проверить, равно ли NULL всё значение строки, без каких-либо дополнительных проверок полей строки.

Логические значения можно также проверить с помощью предикатов

логическое_выражение IS TRUE
логическое_выражение IS NOT TRUE
логическое_выражение IS FALSE
логическое_выражение IS NOT FALSE
логическое_выражение IS UNKNOWN
логическое_выражение IS NOT UNKNOWN

Они всегда возвращают true или false и никогда NULL, даже если какой-любо операнд — NULL. Они интерпретируют значение NULL как «неопределённость». Заметьте, что `IS UNKNOWN` и `IS NOT UNKNOWN` по сути равнозначны `IS NULL` и `IS NOT NULL`, соответственно, за исключением того, что выражение может быть только булевого типа.

Также имеется несколько связанных со сравнениями функций; они перечислены в Таблице 9.3.

Таблица 9.3. Функции сравнения

Функция	Описание	Пример	Результат примера
`num_nonnulls( VARIADIC "any")`	возвращает число аргументов, отличных от NULL	`num_nonnulls(1, NULL, 2)`	2
`num_nulls(VARIADIC "any")`	возвращает число аргументов NULL	`num_nulls(1, NULL, 2)`	1

9.3. Математические функции и операторы

Математические операторы определены для множества типов Postgres Pro. Как работают эти операции с типами, для которых нет стандартных соглашений о математических действиях (например, с типами даты/времени), мы опишем в последующих разделах.

В Таблице 9.4 перечислены все доступные математические операторы.

Таблица 9.4. Математические операторы

Оператор	Описание	Пример	Результат
+	сложение	`2 + 3`	5
-	вычитание	`2 - 3`	-1
*	умножение	`2 * 3`	6
/	деление (при целочисленном делении остаток отбрасывается)	`4 / 2`	2
%	остаток от деления	`5 % 4`	1
^	возведение в степень (вычисляется слева направо)	`2.0 ^ 3.0`	8
|/	квадратный корень	`|/ 25.0`	5
||/	кубический корень	`||/ 27.0`	3
!	факториал (устаревший оператор, его заменяет функция `factorial()`)	`5 !`	120
!!	факториал в префиксной форме (устаревший оператор, его заменяет функция `factorial()`)	`!! 5`	120
@	модуль числа (абсолютное значение)	`@ -5.0`	5
&	битовый AND	`91 & 15`	11
|	битовый OR	`32 | 3`	35

Оператор	Описание	Пример	Результат
#	битовый XOR	17 # 5	20
~	битовый NOT	~1	-2
<<	битовый сдвиг влево	1 << 4	16
>>	битовый сдвиг вправо	8 >> 2	2

Битовые операторы работают только с целочисленными типами данных и с битовыми строками bit и bit varying, как показано в Таблице 9.13.

В Таблице 9.5 перечислены все существующие математические функции. Сокращение dp в ней обозначает тип double precision (плавающее с двойной точностью). Многие из этих функций имеют несколько форм с разными типами аргументов. За исключением случаев, где это указано явно, любая форма функции возвращает результат того же типа, что и аргумент. Функции, работающие с данными double precision, в массе своей используют реализации из системных библиотек сервера, поэтому точность и поведение в граничных случаях может зависеть от системы сервера.

Таблица 9.5. Математические функции

Функция	Тип результата	Описание	Пример	Результат
abs(*x*)	тип аргумента	модуль числа (абсолютное значение)	abs(-17.4)	17.4
cbrt(dp)	dp	кубический корень	cbrt(27.0)	3
ceil(dp или numeric)	тип аргумента	ближайшее целое, большее или равное аргументу	ceil(-42.8)	-42
ceiling(dp или numeric)	тип аргумента	ближайшее целое, большее или равное аргументу (равнозначно ceil)	ceiling(-95.3)	-95
degrees(dp)	dp	преобразование радианов в градусы	degrees(0.5)	28.6478897565412
div(*y* numeric, *x* numeric)	numeric	целочисленный результат *y*/*x*	div(9,4)	2
exp(dp или numeric)	тип аргумента	экспонента	exp(1.0)	2.71828182845905
factorial(bigint)	numeric	факториал	factorial(5)	120
floor(dp или numeric)	тип аргумента	ближайшее целое, меньшее или равное аргументу	floor(-42.8)	-43
ln(dp или numeric)	тип аргумента	натуральный логарифм	ln(2.0)	0.693147180559945
log(dp или numeric)	тип аргумента	логарифм по основанию 10	log(100.0)	2
log(*b* numeric, *x* numeric)	numeric	логарифм по основанию *b*	log(2.0, 64.0)	6.0000000000
mod(*y*, *x*)	зависит от типов аргументов	остаток от деления *y*/*x*	mod(9,4)	1

Функция	Тип результата	Описание	Пример	Результат
pi()	dp	константа «π»	pi()	3.14159265358979
power(*a* dp, *b* dp)	dp	*a* возводится в степень *b*	power(9.0, 3.0)	729
power(*a* numeric, *b* numeric)	numeric	*a* возводится в степень *b*	power(9.0, 3.0)	729
radians(dp)	dp	преобразование градусов в радианы	radians(45.0)	0.785398163397448
round(dp или numeric)	тип аргумента	округление до ближайшего целого	round(42.4)	42
round(*v* numeric, *s* int)	numeric	округление *v* до *s* десятичных знаков	round(42.4382, 2)	42.44
scale(numeric)	integer	масштаб аргумента (число десятичных цифр в дробной части)	scale(8.41)	2
sign(dp или numeric)	тип аргумента	знак аргумента (-1, 0, +1)	sign(-8.4)	-1
sqrt(dp или numeric)	тип аргумента	квадратный корень	sqrt(2.0)	1.4142135623731
trunc(dp или numeric)	тип аргумента	округление к нулю	trunc(42.8)	42
trunc(*v* numeric, *s* int)	numeric	округление к 0 до *s* десятичных знаков	trunc(42.4382, 2)	42.43
width_bucket(*operand* dp, *b1* dp, *b2* dp, *count* int)	int	возвращает номер группы, в которую попадёт *operand* в гистограмме с числом групп *count* равного размера, в диапазоне от *b1* до *b2*; возвращает 0 или *count*+1, если операнд лежит вне диапазона	width_bucket(5.35, 0.024, 10.06, 5)	3
width_bucket(*operand* numeric, *b1* numeric, *b2* numeric, *count* int)	int	возвращает номер группы, в которую попадёт *operand* в гистограмме с числом групп *count* равного размера, в диапазоне от *b1* до *b2*; возвращает 0 или *count*+1, если	width_bucket(5.35, 0.024, 10.06, 5)	3

Функция	Тип результата	Описание	Пример	Результат
		операнд лежит вне диапазона		
width_bucket(*operand* anyelement, *thresholds* anyarray)	int	возвращает номер группы, в которую попадёт *operand* (группы определяются нижними границами, передаваемыми в *thresholds*); возвращает 0, если операнд оказывается левее нижней границы; массив *thresholds* *должен быть отсортирован* по возрастанию, иначе будут получены неожиданные результаты	width_bucket(now(), array['yesterday', 'today', 'tomorrow']::timestamptz[])	2

В Таблице 9.6 перечислены все функции для генерации случайных чисел.

Таблица 9.6. Случайные функции

Функция	Тип результата	Описание
random()	dp	случайное число в диапазоне 0.0 <= x < 1.0
setseed(dp)	void	задаёт отправную точку для последующих вызовов random() (значение между -1.0 и 1.0, включая границы)

Характеристики значений, возвращаемых функцией random() зависят от системы. Для применения в криптографии они непригодны; альтернативы описаны в pgcrypto.

Наконец, в Таблице 9.7 перечислены все имеющиеся тригонометрические функции. Все эти функции принимают аргументы и возвращают значения типа double precision. У каждой функции имеются две вариации: одна измеряет углы в радианах, а вторая — в градусах.

Таблица 9.7. Тригонометрические функции

Функции (в радианах)	Функции (в градусах)	Описание
acos(*x*)	acosd(*x*)	арккосинус
asin(*x*)	asind(*x*)	арксинус
atan(*x*)	atand(*x*)	арктангенс
atan2(*y*, *x*)	atan2d(*y*, *x*)	арктангенс *y*/*x*
cos(*x*)	cosd(*x*)	косинус
cot(*x*)	cotd(*x*)	котангенс
sin(*x*)	sind(*x*)	синус

Функции (в радианах)	Функции (в градусах)	Описание
tan(*x*)	tand(*x*)	тангенс

Примечание

Также можно работать с углами в градусах, применяя вышеупомянутые функции преобразования единиц radians() и degrees(). Однако предпочтительнее использовать тригонометрические функции с градусами, так как это позволяет избежать ошибок округления в особых случаях, например, при вычислении sind(30).

9.4. Строковые функции и операторы

В этом разделе описаны функции и операторы для работы с текстовыми строками. Под строками в данном контексте подразумеваются значения типов character, character varying и text. Если не отмечено обратное, все нижеперечисленные функции работают со всеми этими типами, хотя с типом character следует учитывать возможные эффекты автоматического дополнения строк пробелами. Некоторые из этих функций также поддерживают битовые строки.

В SQL определены несколько строковых функций, в которых аргументы разделяются не запятыми, а ключевыми словами. Они перечислены в Таблице 9.8. Postgres Pro также предоставляет варианты этих функций с синтаксисом, обычным для функций (см. Таблицу 9.9).

Примечание

До версии 8.3 в PostgreSQL эти функции также прозрачно принимали значения некоторых не строковых типов, неявно приводя эти значения к типу text. Сейчас такие приведения исключены, так как они часто приводили к неожиданным результатам. Однако оператор конкатенации строк (||) по-прежнему принимает не только строковые данные, если хотя бы один аргумент имеет строковый тип, как показано в Таблице 9.8. Во всех остальных случаях для повторения предыдущего поведения потребуется добавить явное преобразование в text.

Таблица 9.8. Строковые функции и операторы языка SQL

Функция	Тип результата	Описание	Пример	Результат
string || *string*	text	Конкатенация строк	'Post' || 'greSQL'	PostgreSQL
string || *не string* или *не string* || *string*	text	Конкатенация строк с одним не строковым операндом	'Value: ' || 42	Value: 42
bit_length(*string*)	int	Число бит в строке	bit_length('jose')	32
char_length(*string*) или character_ length(*string*)	int	Число символов в строке	char_length('jose')	4
lower(*string*)	text	Переводит символы строки в нижний регистр	lower('TOM')	tom
octet_length(*string*)	int	Число байт в строке	octet_length('jose')	4

Функция	Тип результата	Описание	Пример	Результат
overlay(*string* placing *string* from int [for int])	text	Заменяет подстроку	overlay('Txxxxas' placing 'hom' from 2 for 4)	Thomas
position(*substring* in *string*)	int	Положение указанной подстроки	position('om' in 'Thomas')	3
substring(*string* [from int] [for int])	text	Извлекает подстроку	substring('Thomas' from 2 for 3)	hom
substring(*string* from *шаблон*)	text	Извлекает подстроку, соответствующую регулярному выражению в стиле POSIX. Подробно шаблоны описаны в Разделе 9.7.	substring('Thomas' from '...$')	mas
substring(*string* from *шаблон* for *спецсимвол*)	text	Извлекает подстроку, соответствующую регулярному выражению в стиле SQL. Подробно шаблоны описаны в Разделе 9.7.	substring('Thomas' from '%#"o_a#"_' for '#')	oma
trim([leading | trailing | both] [*characters*] from *string*)	text	Удаляет наибольшую подстроку, содержащую только символы *characters* (по умолчанию пробелы), с начала (leading), с конца (trailing) или с обеих сторон (both, (по умолчанию)) строки *string*	trim(both 'xyz' from 'yxTomxx')	Tom
trim([leading | trailing | both] [from] *string* [, *characters*])	text	Нестандартный синтаксис trim()	trim(both from 'yxTomxx', 'xyz')	Tom
upper(*string*)	text	Переводит символы строки в верхний регистр	upper('tom')	TOM

Кроме этого, в PostgreSQL есть и другие функции для работы со строками, перечисленные в Таблице 9.9. Некоторые из них используются в качестве внутренней реализации стандартных строковых функций SQL, приведённых в Таблице 9.8.

Таблица 9.9. Другие строковые функции

Функция	Тип результата	Описание	Пример	Результат
`ascii( string)`	`int`	Возвращает ASCII-код первого символа аргумента. Для UTF8 возвращает код символа в Unicode. Для других многобайтных кодировок аргумент должен быть ASCII-символом.	`ascii('x')`	`120`
`btrim( string text [, characters text])`	`text`	Удаляет наибольшую подстроку, состоящую только из символов *characters* (по умолчанию пробелов), с начала и с конца строки *string*	`btrim( 'xyxtrimyyx', 'xyz')`	`trim`
`chr( int)`	`text`	Возвращает символ с данным кодом. Для UTF8 аргумент воспринимается как код символа Unicode, а для других кодировок он должен указывать на ASCII-символ. Код 0 (NULL) не допускается, так как байты с нулевым кодом в текстовых строках сохранить нельзя.	`chr(65)`	`A`
`concat( str "any" [, str "any" [, ...] ])`	`text`	Соединяет текстовые представления всех аргументов, игнорируя NULL.	`concat('abcde', 2, NULL, 22)`	`abcde222`
`concat_ws( sep text, str "any" [, str "any" [, ...] ])`	`text`	Соединяет все аргументы, кроме первого, через разделитель, игнорируя аргументы NULL. Разделитель	`concat_ws(', ', 'abcde', 2, NULL, 22)`	`abcde,2,22`

Функция	Тип результата	Описание	Пример	Результат
		указывается в первом аргументе.		
convert(*string* bytea, *src_encoding* name, *dest_encoding* name)	bytea	Преобразует строку *string* из кодировки *src_encoding* в *dest_encoding* . Переданная строка должна быть допустимой для исходной кодировки. Преобразования могут быть определены с помощью CREATE CONVERSION. Все встроенные преобразования перечислены в Таблице 9.10.	convert('text_in_utf8', 'UTF8', 'LATIN1')	строка text_in_utf8 , представленная в кодировке Latin-1 (ISO 8859-1)
convert_from(*string* bytea, *src_encoding* name)	text	Преобразует строку *string* из кодировки *src_encoding* в кодировку базы данных. Переданная строка должна быть допустимой для исходной кодировки.	convert_from('text_in_utf8', 'UTF8')	строка text_in_utf8 , представленная в кодировке текущей базы данных
convert_to(*string* text, *dest_encoding* name)	bytea	Преобразует строку в кодировку *dest_encoding* .	convert_to('некоторый текст', 'UTF8')	некоторый текст, представленный в кодировке UTF8
decode(*string* text, *format* text)	bytea	Получает двоичные данные из текстового представления в *string*. Значения параметра *format* те же, что и для функции encode.	decode('MTIzAAE=', 'base64')	\x3132330001
encode(*data* bytea, *format* text)	text	Переводит двоичные данные в текстовое представление в одном из форматов: base64, hex, escape. Формат escape преобразует нулевые байты	encode('123\000\001', 'base64')	MTIzAAE=

Функция	Тип результата	Описание	Пример	Результат
		и байты с 1 в старшем бите в восьмеричные последовательности `\nnn` и дублирует обратную косую черту.		
`format( *formatstr* text [, *formatarg* "any" [, ...] ])`	`text`	Форматирует аргумент в соответствии со строкой формата. Эта функция работает подобно `sprintf` в языке C. См. Подраздел 9.4.1.	`format('Hello %s, %1$s', 'World')`	`Hello World, World`
`initcap( *string*)`	`text`	Переводит первую букву каждого слова в строке в верхний регистр, а остальные — в нижний. Словами считаются последовательности алфавитно-цифровых символов, разделённые любыми другими символами.	`initcap('hi THOMAS')`	`Hi Thomas`
`left( *str* text, *n* int)`	`text`	Возвращает первые *n* символов в строке. Когда *n* меньше нуля, возвращаются все символы слева, кроме последних |*n*|.	`left('abcde', 2)`	`ab`
`length( *string*)`	`int`	Число символов в строке *string*	`length('jose')`	`4`
`length( *string* bytea, *encoding* name )`	`int`	Число символов, которые содержит строка *string* в заданной кодировке *encoding*. Переданная строка должна быть допустимой в этой кодировке.	`length('jose', 'UTF8')`	`4`
`lpad( *string* text, *length* int [, *fill* text])`	`text`	Дополняет строку *string* слева до длины *length* символами *fill*	`lpad('hi', 5, 'xy')`	`xyxhi`

Функция	**Тип результата**	**Описание**	**Пример**	**Результат**
		(по умолчанию пробелами). Если длина строки уже больше заданной, она обрезается справа.		
ltrim(*string* text [, *characters* text])	text	Удаляет наибольшую подстроку, содержащую только символы *characters* (по умолчанию пробелы), с начала строки *string*	ltrim('zzzytest', 'xyz')	test
md5(*string*)	text	Вычисляет MD5-хеш строки *string* и возвращает результат в 16-ричном виде	md5('abc')	90015098 3cd24fb0 d6963f7d 28e17f72
parse_ident(*qualified_identifier* text [, *strictmode* boolean DEFAULT true])	text[]	Раскладывает полный идентификатор, задаваемый параметром *qualified_identifier* , на массив идентификаторов, удаляя кавычки, обрамляющие отдельные идентификаторы. По умолчанию лишние символы после последнего идентификатора вызывают ошибку, но если отключить строгий режим (передать во втором параметре false), такие символы игнорируются. (Это поведение полезно для разбора имён таких объектов, как функции.) Заметьте, что эта функция не усекает чрезмерно	parse_ident('"SomeSchema".someTable')	{SomeSchema, someTable}

Функция	Тип результата	Описание	Пример	Результат
		длинные идентификаторы. Если вы хотите получить усечённые имена, можно привести результат к name[].		
pg_client_encoding()	name	Возвращает имя текущей клиентской кодировки	pg_client_encoding()	SQL_ASCII
quote_ident(*string* text)	text	Переданная строка оформляется для использования в качестве идентификатора в SQL -операторе. При необходимости идентификатор заключается в кавычки (например, если он содержит символы, недопустимые в открытом виде, или буквы в разном регистре). Если переданная строка содержит кавычки, они дублируются. См. также Пример 40.1.	quote_ident('Foo bar')	"Foo bar"
quote_literal(*string* text)	text	Переданная строка оформляется для использования в качестве текстовой строки в SQL-операторе. Включённые символы апостроф и обратная косая черта при этом дублируются. Заметьте, что quote_literal возвращает NULL, когда на вход ей передаётся строка NULL; если же	quote_literal(E'O\'Reilly')	'O''Reilly'

Функция	Тип результата	Описание	Пример	Результат
		нужно получить представление и такого аргумента, лучше использовать quote_nullable . См. также Пример 40.1.		
quote_literal(*value* anyelement)	text	Переводит данное значение в текстовый вид и заключает в апострофы как текстовую строку. Символы апостроф и обратная косая черта при этом дублируются.	quote_literal(42.5)	'42.5'
quote_nullable(*string* text)	text	Переданная строка оформляется для использования в качестве текстовой строки в SQL-операторе; при этом для аргумента NULL возвращается строка NULL. Символы апостроф и обратная косая черта дублируются должным образом. См. также Пример 40.1.	quote_nullable(NULL)	NULL
quote_nullable(*value* anyelement)	text	Переводит данное значение в текстовый вид и заключает в апострофы как текстовую строку, при этом для аргумента NULL возвращается строка NULL. Символы апостроф и обратная косая черта дублируются должным образом.	quote_nullable(42.5)	'42.5'
regexp_match(*string* text, *pattern* text [, *flags* text])	text[]	Возвращает подходящие подстроки, полученные из	regexp_match('foobar bequebaz', '(bar)(beque)')	{bar,beque}

Функция	Тип результата	Описание	Пример	Результат
		первого вхождения регулярного выражения POSIX в строке *string*. Подробности описаны в Подразделе 9.7.3.		
`regexp_matches ( string text, pattern text [, flags text])`	`setof text[]`	Возвращает подходящие подстроки, полученные в результате применения регулярного выражения POSIX к *string*. Подробности описаны в Подразделе 9.7.3.	`regexp_matches( 'foobar bequebaz', 'ba.', 'g')`	`{bar}` `{baz}` (2 строки)
`regexp_replace ( string text, pattern text, замена text [, flags text])`	`text`	Заменяет подстроки, соответствующие заданному регулярному выражению в стиле POSIX. Подробности описаны в Подразделе 9.7.3.	`regexp_replace( 'Thomas', '. [mN]a.', 'M')`	`ThM`
`regexp_split_ to_array( string text, pattern text [, flags text ])`	`text[]`	Разделяет содержимое *string* на элементы, используя в качестве разделителя регулярное выражение POSIX. Подробности описаны в Подразделе 9.7.3.	`regexp_split_ to_array( 'hello world', '\s+')`	`{hello,world}`
`regexp_split_ to_table( string text, pattern text [, flags text])`	`setof text`	Разделяет содержимое *string* на элементы, используя в качестве разделителя регулярное выражение POSIX. Подробности описаны в Подразделе 9.7.3.	`regexp_split_ to_table( 'hello world', '\s+')`	`hello` `world` (2 строки)

Функция	**Тип результата**	**Описание**	**Пример**	**Результат**
`repeat( string text, number int)`	`text`	Повторяет содержимое *string* указанное число (*number*) раз	`repeat('Pg', 4)`	`PgPgPgPg`
`replace( string text, from text, to text)`	`text`	Заменяет все вхождения в *string* подстроки *from* подстрокой *to*	`replace( 'abcdefabcdef', 'cd', 'XX')`	`abXXefabXXef`
`reverse( str)`	`text`	Возвращает перевёрнутую строку	`reverse( 'abcde')`	`edcba`
`right( str text, n int)`	`text`	Возвращает последние *n* символов в строке. Когда *n* меньше нуля, возвращаются все символы справа, кроме первых |*n*|.	`right('abcde', 2)`	`de`
`rpad( string text, length int [, fill text])`	`text`	Дополняет строку *string* справа до длины *length* символами *fill* (по умолчанию пробелами). Если длина строки уже больше заданной, она обрезается.	`rpad('hi', 5, 'xy')`	`hixyx`
`rtrim( string text [, characters text])`	`text`	Удаляет наибольшую подстроку, содержащую только символы *characters* (по умолчанию пробелы), с конца строки *string*	`rtrim( 'testxxzx', 'xyz')`	`test`
`split_part( string text, delimiter text, field int)`	`text`	Разделяет строку *string* по символу *delimiter* и возвращает элемент по заданному номеру (считая с 1)	`split_part( 'abc~@~def ~@~ghi', '~@~', 2)`	`def`
`strpos( string, substring)`	`int`	Возвращает положение указанной подстроки (подобно `position( substring in string)`, но с	`strpos('high', 'ig')`	`2`

Функция	Тип результата	Описание	Пример	Результат
		другим порядком аргументов)		
substr(*string*, *from* [, *count*])	text	Извлекает подстроку (подобно substring(*string* from *from* for *count*))	substr('alphabet', 3, 2)	ph
to_ascii(*string* text [, *encoding* text])	text	Преобразует *string* в ASCII из кодировки *encoding* (поддерживаются только LATIN1, LATIN2, LATIN9 и WIN1250)	to_ascii('Karel')	Karel
to_hex(*number* int или bigint)	text	Преобразует число *number* в 16-ричный вид	to_hex(2147483647)	7fffffff
translate(*string* text, *from* text, *to* text)	text	Заменяет символы в *string*, найденные в наборе *from*, на соответствующие символы в множестве *to*. Если строка *from* длиннее *to*, найденные в исходной строке лишние символы *from* удаляются.	translate('12345', '143', 'ax')	a2x5

Функции concat, concat_ws и format принимают переменное число аргументов, так что им для объединения или форматирования можно передавать значения в виде массива, помеченного ключевым словом VARIADIC (см. Подраздел 35.4.5). Элементы такого массива обрабатываются, как если бы они были обычными аргументами функции. Если вместо массива в соответствующем аргументе передаётся NULL, функции concat и concat_ws возвращают NULL, а format воспринимает NULL как массив нулевого размера.

См. также агрегатную функцию string_agg в Разделе 9.20.

Таблица 9.10. Встроенные преобразования

Имя преобразования [a]	Исходная кодировка	Целевая кодировка
ascii_to_mic	SQL_ASCII	MULE_INTERNAL
ascii_to_utf8	SQL_ASCII	UTF8
big5_to_euc_tw	BIG5	EUC_TW
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf8	BIG5	UTF8
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf8	EUC_CN	UTF8

Имя преобразования [a]	**Исходная кодировка**	**Целевая кодировка**
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS
euc_jp_to_utf8	EUC_JP	UTF8
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf8	EUC_KR	UTF8
euc_tw_to_big5	EUC_TW	BIG5
euc_tw_to_mic	EUC_TW	MULE_INTERNAL
euc_tw_to_utf8	EUC_TW	UTF8
gb18030_to_utf8	GB18030	UTF8
gbk_to_utf8	GBK	UTF8
iso_8859_10_to_utf8	LATIN6	UTF8
iso_8859_13_to_utf8	LATIN7	UTF8
iso_8859_14_to_utf8	LATIN8	UTF8
iso_8859_15_to_utf8	LATIN9	UTF8
iso_8859_16_to_utf8	LATIN10	UTF8
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf8	LATIN1	UTF8
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf8	LATIN2	UTF8
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL
iso_8859_3_to_utf8	LATIN3	UTF8
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL
iso_8859_4_to_utf8	LATIN4	UTF8
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8R
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf8	ISO_8859_5	UTF8
iso_8859_5_to_windows_1251	ISO_8859_5	WIN1251
iso_8859_5_to_windows_866	ISO_8859_5	WIN866
iso_8859_6_to_utf8	ISO_8859_6	UTF8
iso_8859_7_to_utf8	ISO_8859_7	UTF8
iso_8859_8_to_utf8	ISO_8859_8	UTF8
iso_8859_9_to_utf8	LATIN5	UTF8
johab_to_utf8	JOHAB	UTF8
koi8_r_to_iso_8859_5	KOI8R	ISO_8859_5
koi8_r_to_mic	KOI8R	MULE_INTERNAL
koi8_r_to_utf8	KOI8R	UTF8
koi8_r_to_windows_1251	KOI8R	WIN1251

Имя преобразования [a]	Исходная кодировка	Целевая кодировка
koi8_r_to_windows_866	KOI8R	WIN866
koi8_u_to_utf8	KOI8U	UTF8
mic_to_ascii	MULE_INTERNAL	SQL_ASCII
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8R
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN1251
mic_to_windows_866	MULE_INTERNAL	WIN866
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf8	SJIS	UTF8
windows_1258_to_utf8	WIN1258	UTF8
uhc_to_utf8	UHC	UTF8
utf8_to_ascii	UTF8	SQL_ASCII
utf8_to_big5	UTF8	BIG5
utf8_to_euc_cn	UTF8	EUC_CN
utf8_to_euc_jp	UTF8	EUC_JP
utf8_to_euc_kr	UTF8	EUC_KR
utf8_to_euc_tw	UTF8	EUC_TW
utf8_to_gb18030	UTF8	GB18030
utf8_to_gbk	UTF8	GBK
utf8_to_iso_8859_1	UTF8	LATIN1
utf8_to_iso_8859_10	UTF8	LATIN6
utf8_to_iso_8859_13	UTF8	LATIN7
utf8_to_iso_8859_14	UTF8	LATIN8
utf8_to_iso_8859_15	UTF8	LATIN9
utf8_to_iso_8859_16	UTF8	LATIN10
utf8_to_iso_8859_2	UTF8	LATIN2
utf8_to_iso_8859_3	UTF8	LATIN3
utf8_to_iso_8859_4	UTF8	LATIN4

Имя преобразования [a]	Исходная кодировка	Целевая кодировка
utf8_to_iso_8859_5	UTF8	ISO_8859_5
utf8_to_iso_8859_6	UTF8	ISO_8859_6
utf8_to_iso_8859_7	UTF8	ISO_8859_7
utf8_to_iso_8859_8	UTF8	ISO_8859_8
utf8_to_iso_8859_9	UTF8	LATIN5
utf8_to_johab	UTF8	JOHAB
utf8_to_koi8_r	UTF8	KOI8R
utf8_to_koi8_u	UTF8	KOI8U
utf8_to_sjis	UTF8	SJIS
utf8_to_windows_1258	UTF8	WIN1258
utf8_to_uhc	UTF8	UHC
utf8_to_windows_1250	UTF8	WIN1250
utf8_to_windows_1251	UTF8	WIN1251
utf8_to_windows_1252	UTF8	WIN1252
utf8_to_windows_1253	UTF8	WIN1253
utf8_to_windows_1254	UTF8	WIN1254
utf8_to_windows_1255	UTF8	WIN1255
utf8_to_windows_1256	UTF8	WIN1256
utf8_to_windows_1257	UTF8	WIN1257
utf8_to_windows_866	UTF8	WIN866
utf8_to_windows_874	UTF8	WIN874
windows_1250_to_iso_ 8859_2	WIN1250	LATIN2
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf8	WIN1250	UTF8
windows_1251_to_iso_ 8859_5	WIN1251	ISO_8859_5
windows_1251_to_koi8_r	WIN1251	KOI8R
windows_1251_to_mic	WIN1251	MULE_INTERNAL
windows_1251_to_utf8	WIN1251	UTF8
windows_1251_to_windows_ 866	WIN1251	WIN866
windows_1252_to_utf8	WIN1252	UTF8
windows_1256_to_utf8	WIN1256	UTF8
windows_866_to_iso_8859_ 5	WIN866	ISO_8859_5
windows_866_to_koi8_r	WIN866	KOI8R
windows_866_to_mic	WIN866	MULE_INTERNAL
windows_866_to_utf8	WIN866	UTF8
windows_866_to_windows_ 1251	WIN866	WIN

Имя преобразования [a]	Исходная кодировка	Целевая кодировка
windows_874_to_utf8	WIN874	UTF8
euc_jis_2004_to_utf8	EUC_JIS_2004	UTF8
utf8_to_euc_jis_2004	UTF8	EUC_JIS_2004
shift_jis_2004_to_utf8	SHIFT_JIS_2004	UTF8
utf8_to_shift_jis_2004	UTF8	SHIFT_JIS_2004
euc_jis_2004_to_shift_jis_2004	EUC_JIS_2004	SHIFT_JIS_2004
shift_jis_2004_to_euc_jis_2004	SHIFT_JIS_2004	EUC_JIS_2004

[a]Имена преобразований следуют стандартной схеме именования. К официальному названию исходной кодировки, в котором все не алфавитно-цифровые символы заменяются подчёркиваниями, добавляется _to_, а за ним аналогично подготовленное имя целевой кодировки. Таким образом, имена кодировок могут не совпадать буквально с общепринятыми названиями.

9.4.1. format

Функция format выдаёт текст, отформатированный в соответствии со строкой формата, подобно функции sprintf в C.

format(*formatstr* text [, *formatarg* "any" [, ...]])

formatstr — строка, определяющая, как будет форматироваться результат. Обычный текст в строке формата непосредственно копируется в результат, за исключением *спецификаторов формата*. Спецификаторы формата представляют собой местозаполнители, определяющие, как должны форматироваться и выводиться в результате аргументы функции. Каждый аргумент *formatarg* преобразуется в текст по правилам выводам своего типа данных, а затем форматируется и вставляется в результирующую строку согласно спецификаторам формата.

Спецификаторы формата предваряются символом % и имеют форму

%[*позиция*][*флаги*][*ширина*]*тип*

Здесь:

позиция (необязателен)

Строка вида *n*$, где *n* — индекс выводимого аргумента. Индекс, равный 1, выбирает первый аргумент после *formatstr*. Если *позиция* опускается, по умолчанию используется следующий аргумент по порядку.

флаги (необязателен)

Дополнительные параметры, управляющие форматированием данного спецификатора. В настоящее время поддерживается только знак минус (-), который выравнивает результата спецификатора по левому краю. Он работает, только если также определена *ширина*.

ширина (необязателен)

Задаёт *минимальное* число символов, которое будет занимать результат данного спецификатора. Выводимое значение выравнивается по правой или левой стороне (в зависимости от флага -) с дополнением необходимым числом пробелов. Если ширина слишком мала, она просто игнорируется, т. е. результат не усекается. Ширину можно обозначить положительным целым, звёздочкой (*), тогда ширина будет получена из следующего аргумента функции, или строкой вида **n*$, тогда ширина будет задаваться в *n*-ом аргументе функции.

Если ширина передаётся в аргументе функции, этот аргумент выбирается до аргумента, используемого для спецификатора. Если аргумент ширины отрицательный, результат выравнивается по левой стороне (как если бы был указан флаг -) в рамках поля длины abs(*ширина*).

тип (обязателен)

Тип спецификатора определяет преобразование соответствующего выводимого значения. Поддерживаются следующие типы:

- `s` форматирует значение аргумента как простую строку. Значение NULL представляется пустой строкой.
- `I` обрабатывает значение аргумента как SQL-идентификатор, при необходимости заключая его в кавычки. Значение NULL для такого преобразования считается ошибочным (так же, как и для `quote_ident`).
- `L` заключает значение аргумента в апострофы, как строку SQL. Значение NULL выводится буквально, как `NULL`, без кавычек (так же, как и с `quote_nullable`).

В дополнение к спецификаторам, описанным выше, можно использовать спецпоследовательность `%%`, которая просто выведет символ `%`.

Несколько примеров простых преобразований формата:

```
SELECT format('Hello %s', 'World');
Результат: Hello World

SELECT format('Testing %s, %s, %s, %%', 'one', 'two', 'three');
Результат: Testing one, two, three, %

SELECT format('INSERT INTO %I VALUES(%L)', 'Foo bar', E'O\'Reilly');
Результат: INSERT INTO "Foo bar" VALUES('O''Reilly')

SELECT format('INSERT INTO %I VALUES(%L)', 'locations', 'C:\Program Files');
Результат: INSERT INTO locations VALUES('C:\Program Files')
```

Следующие примеры иллюстрируют использование поля *ширина* и флага `-`:

```
SELECT format('|%10s|', 'foo');
Результат: |       foo|

SELECT format('|%-10s|', 'foo');
Результат: |foo       |

SELECT format('|%*s|', 10, 'foo');
Результат: |       foo|

SELECT format('|%*s|', -10, 'foo');
Результат: |foo       |

SELECT format('|%-*s|', 10, 'foo');
Результат: |foo       |

SELECT format('|%-*s|', -10, 'foo');
Результат: |foo       |
```

Эти примеры показывают применение полей *позиция*:

```
SELECT format('Testing %3$s, %2$s, %1$s', 'one', 'two', 'three');
Результат: Testing three, two, one

SELECT format('|%*2$s|', 'foo', 10, 'bar');
Результат: |       bar|

SELECT format('|%1$*2$s|', 'foo', 10, 'bar');
```

Результат: | foo|

В отличие от стандартной функции C `sprintf`, функция `format` в Postgres Pro позволяет комбинировать в одной строке спецификаторы с полями *позиция* и без них. Спецификатор формата без поля *позиция* всегда использует следующий аргумент после последнего выбранного. Кроме того, функция `format` не требует, чтобы в строке формата использовались все аргументы функции. Пример этого поведения:

```
SELECT format('Testing %3$s, %2$s, %s', 'one', 'two', 'three');
Результат: Testing three, two, three
```

Спецификаторы формата `%I` и `%L` особенно полезны для безопасного составления динамических операторов SQL. См. Пример 40.1.

9.5. Функции и операторы двоичных строк

В этом разделе описываются функции и операторы для работы с данными типа `bytea`.

В SQL определены несколько строковых функций, в которых аргументы разделяются не запятыми, а ключевыми словами. Подробнее это описано в Таблице 9.11. Postgres Pro также предоставляет варианты этих функций с синтаксисом, обычным для функций (см. Таблицу 9.12).

> **Примечание**
>
> В примерах, приведённых на этой странице, подразумевается, что параметр сервера `bytea_output` равен `escape` (выбран традиционный формат Postgres Pro).

Таблица 9.11. SQL-функции и операторы для работы с двоичными строками

Функция	Тип результата	Описание	Пример	Результат
string `||` *string*	bytea	Конкатенация строк	`'\\Post'::bytea || '\047gres \000'::bytea`	`\\Post'gres\000`
`octet_length(` *string*`)`	int	Число байт в двоичной строке	`octet_length( 'jo \000se'::bytea)`	5
`overlay(` *string* `placing` *string* `from int [for int])`	bytea	Заменяет подстроку	`overlay( 'Th \000omas'::bytea placing '\002\003'::bytea from 2 for 3)`	`T\\002\\003mas`
`position(` *substring* `in` *string*`)`	int	Положение указанной подстроки	`position( '\000om'::bytea in 'Th \000omas'::bytea)`	3
`substring(` *string* `[from int] [for int])`	bytea	Извлекает подстроку	`substring( 'Th \000omas'::bytea from 2 for 3)`	`h\000o`
`trim([both]` *bytes* `from` *string*`)`	bytea	Удаляет наибольшую строку, содержащую только байты,	`trim( '\000\001'::bytea from '\000Tom \001'::bytea)`	Tom

Функция	Тип результата	Описание	Пример	Результат
		заданные в параметре *bytes*, с начала и с конца строки *string*		

В PostgreSQL есть и другие функции для работы с двоичными строками, перечисленные в Таблице 9.12. Некоторые из них используются в качестве внутренней реализации стандартных функций SQL, приведённых в Таблице 9.11.

Таблица 9.12. Другие функции для работы с двоичными строками

Функция	Тип результата	Описание	Пример	Результат
`btrim( *string* bytea, *bytes* bytea)`	`bytea`	Удаляет наибольшую строку, содержащую только байты, заданные в параметре *bytes*, с начала и с конца строки *string*	`btrim( '\000trim \001'::bytea, '\000\001'::bytea)`	`trim`
`decode( *string* text, *format* text)`	`bytea`	Получает двоичные данные из текстового представления в *string*. Значения параметра *format* те же, что и для функции `encode`.	`decode( '123\000456', 'escape')`	`123\000456`
`encode( *data* bytea, *format* text)`	`text`	Переводит двоичные данные в текстовое представление в одном из форматов: `base64`, `hex`, `escape`. Формат `escape` преобразует нулевые байты и байты с 1 в старшем бите в восьмеричные последовательности `\nnn` и дублирует обратную косую черту.	`encode( '123\000456'::bytea, 'escape')`	`123\000456`
`get_bit( *string*, *offset*)`	`int`	Извлекает бит из строки	`get_bit( 'Th \000omas'::bytea, 45)`	`1`
`get_byte( *string*, *offset*)`	`int`	Извлекает байт из строки	`get_byte( 'Th \000omas'::bytea, 4)`	`109`

Функция	Тип результата	Описание	Пример	Результат
`length( string)`	`int`	Длина двоичной строки	`length( 'jo \000se'::bytea)`	`5`
`md5( string)`	`text`	Вычисляет MD5-хеш строки *string* и возвращает результат в 16-ричном виде	`md5( 'Th \000omas'::bytea)`	`8ab2d3c9 689aaf18 b4958c33 4c82d8b1`
`set_bit( string, offset, newvalue)`	`bytea`	Устанавливает значение бита в строке	`set_bit( 'Th \000omas'::bytea, 45, 0)`	`Th\000omAs`
`set_byte( string, offset, newvalue)`	`bytea`	Устанавливает значение байта в строке	`set_byte( 'Th \000omas'::bytea, 4, 64)`	`Th\000o@as`

Для функций `get_byte` и `set_byte` байты нумеруется с 0. Функции `get_bit` и `set_bit` нумеруют биты справа налево; например, бит 0 будет меньшим значащим битом первого байта, а бит 15 — большим значащим битом второго байта.

См. также агрегатную функцию `string_agg` в Разделе 9.20 и функции для работы с большими объектами в Разделе 32.4.

9.6. Функции и операторы для работы с битовыми строками

В этом разделе описываются функции и операторы, предназначенные для работы с битовыми строками, то есть с данными типов `bit` и `bit varying`. Помимо обычных операторов сравнения, с такими данными можно использовать операторы, перечисленные в Таблице 9.13. Заметьте, что операторы `&`, `|` и `#` работают только с двоичными строками одинаковой длины. Операторы побитового сдвига сохраняют длины исходных строк, как показано в примерах.

Таблица 9.13. Операторы для работы с битовыми строками

Оператор	Описание	Пример	Результат
`||`	конкатенация	`B'10001' || B'011'`	`10001011`
`&`	битовый AND	`B'10001' & B'01101'`	`00001`
`|`	битовый OR	`B'10001' | B'01101'`	`11101`
`#`	битовый XOR	`B'10001' # B'01101'`	`11100`
`~`	битовый NOT	`~ B'10001'`	`01110`
`<<`	битовый сдвиг влево	`B'10001' << 3`	`01000`
`>>`	битовый сдвиг вправо	`B'10001' >> 2`	`00100`

Следующие функции языка SQL работают как с символьными, так и с битовыми строками: `length`, `bit_length`, `octet_length`, `position`, `substring`, `overlay`.

С битовыми и двоичными строками работают функции `get_bit` и `set_bit`. При работе с битовыми строками эти функции нумеруют биты слева направо и самый левый бит считается нулевым.

Кроме того, целые значения можно преобразовать в тип `bit` и обратно. Например:

```
44::bit(10)                    0000101100
```

```
44::bit(3)                  100
cast(-44 as bit(12))        111111010100
'1110'::bit(4)::integer     14
```

Заметьте, что приведение к типу «bit» без длины будет означать приведение к `bit(1)`, и в результате будет получен только один менее значащий бит числа.

> **Примечание**
>
> Приведение целого числа к типу `bit(n)` копирует правые `n` бит числа. Если же целое преобразуется в битовую строку большей длины, чем требуется для этого числа, она дополняется слева битами знака числа.

9.7. Поиск по шаблону

Postgres Pro предлагает три разных способа поиска текста по шаблону: традиционный оператор `LIKE` языка SQL, более современный `SIMILAR TO` (добавленный в SQL:1999) и регулярные выражения в стиле POSIX. Помимо простых операторов, отвечающих на вопрос «соответствует ли строка этому шаблону?», в Postgres Pro есть функции для извлечения или замены соответствующих подстрок и для разделения строки по заданному шаблону.

> **Подсказка**
>
> Если этих встроенных возможностей оказывается недостаточно, вы можете написать собственные функции на языке Perl или Tcl.

> **Внимание**
>
> Хотя чаще всего поиск по регулярному выражению бывает очень быстрым, регулярные выражения бывают и настолько сложными, что их обработка может занять приличное время и объём памяти. Поэтому опасайтесь шаблонов регулярных выражений, поступающих из недоверенных источников. Если у вас нет другого выхода, рекомендуется ввести тайм-аут для операторов.
>
> Поиск с шаблонами `SIMILAR TO` несёт те же риски безопасности, так как конструкция `SIMILAR TO` предоставляет во многом те же возможности, что и регулярные выражения в стиле POSIX.
>
> Поиск с `LIKE` гораздо проще, чем два другие варианта, поэтому его безопаснее использовать с недоверенными источниками шаблонов поиска.

9.7.1. `LIKE`

строка LIKE *шаблон* [ESCAPE *спецсимвол*]
строка NOT LIKE *шаблон* [ESCAPE *спецсимвол*]

Выражение `LIKE` возвращает true, если *строка* соответствует заданному *шаблону*. (Как можно было ожидать, выражение `NOT LIKE` возвращает false, когда `LIKE` возвращает true, и наоборот. Этому выражению равносильно выражение `NOT (`*строка* `LIKE` *шаблон*`)`.)

Если *шаблон* не содержит знаков процента и подчёркиваний, тогда шаблон представляет в точности строку и `LIKE` работает как оператор сравнения. Подчёркивание (_) в *шаблоне* подменяет (вместо него подходит) любой символ; а знак процента (%) подменяет любую (в том числе и пустую) последовательность символов.

Несколько примеров:

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

При проверке по шаблону LIKE всегда рассматривается вся строка. Поэтому, если нужно найти последовательность символов где-то в середине строки, шаблон должен начинаться и заканчиваться знаками процента.

Чтобы найти в строке буквальное вхождение знака процента или подчёркивания, перед соответствующим символом в *шаблоне* нужно добавить спецсимвол. По умолчанию в качестве спецсимвола выбрана обратная косая черта, но с помощью предложения ESCAPE можно выбрать и другой. Чтобы включить спецсимвол в шаблон поиска, продублируйте его.

> **Примечание**
>
> Если параметр standard_conforming_strings выключен, каждый символ обратной косой черты, записываемый в текстовой константе, нужно дублировать. Подробнее это описано в Подразделе 4.1.2.1.

Также можно отказаться от спецсимвола, написав ESCAPE ''. При этом механизм спецпоследовательностей фактически отключается и использовать знаки процента и подчёркивания буквально в шаблоне нельзя.

Вместо LIKE можно использовать ключевое слово ILIKE, чтобы поиск был регистр-независимым с учётом текущей языковой среды. Этот оператор не описан в стандарте SQL; это расширение Postgres Pro.

Кроме того, в Postgres Pro есть оператор ~~, равнозначный LIKE, и ~~*, соответствующий ILIKE. Есть также два оператора !~~ и !~~*, представляющие NOT LIKE и NOT ILIKE, соответственно. Все эти операторы относятся к особенностям Postgres Pro. Вы можете увидеть их, например, в выводе команды EXPLAIN, так как при разборе запроса проверка LIKE и подобные заменяются ими.

Фразы LIKE, ILIKE, NOT LIKE и NOT ILIKE в синтаксисе Postgres Pro обычно обрабатываются как операторы; например, их можно использовать в конструкциях *выражение оператор* ANY (*подвыражение*), хотя предложение ESCAPE здесь добавить нельзя. В некоторых особых случаях всё же может потребоваться использовать вместо них нижележащие операторы.

9.7.2. Регулярные выражения SIMILAR TO

строка SIMILAR TO *шаблон* [ESCAPE *спецсимвол*]
строка NOT SIMILAR TO *шаблон* [ESCAPE *спецсимвол*]

Оператор SIMILAR TO возвращает true или false в зависимости от того, соответствует ли данная строка шаблону или нет. Он работает подобно оператору LIKE, только его шаблоны соответствуют определению регулярных выражений в стандарте SQL. Регулярные выражения SQL представляют собой любопытный гибрид синтаксиса LIKE с синтаксисом обычных регулярных выражений.

Как и LIKE, условие SIMILAR TO истинно, только если шаблон соответствует всей строке; это отличается от условий с регулярными выражениями, в которых шаблон может соответствовать любой части строки. Также подобно LIKE, SIMILAR TO воспринимает символы _ и % как знаки подстановки, подменяющие любой один символ или любую подстроку, соответственно (в регулярных выражениях POSIX им аналогичны символы . и .*).

Помимо средств описания шаблонов, позаимствованных от LIKE, SIMILAR TO поддерживает следующие метасимволы, унаследованные от регулярных выражений POSIX:

- `|` означает выбор (одного из двух вариантов).
- `*` означает повторение предыдущего элемента 0 и более раз.
- `+` означает повторение предыдущего элемента 1 и более раз.
- `?` означает вхождение предыдущего элемента 0 или 1 раз.
- `{`*m*`}` означает повторяет предыдущего элемента ровно *m* раз.
- `{`*m*`,}` означает повторение предыдущего элемента *m* или более раз.
- `{`*m*`,`*n*`}` означает повторение предыдущего элемента не менее чем *m* и не более чем *n* раз.
- Скобки `()` объединяют несколько элементов в одну логическую группу.
- Квадратные скобки `[...]` обозначают класс символов так же, как и в регулярных выражениях POSIX.

Обратите внимание, точка (`.`) не является метасимволом для оператора `SIMILAR TO`.

Как и с `LIKE`, обратная косая черта отменяет специальное значение любого из этих метасимволов, а предложение `ESCAPE` позволяет выбрать другой спецсимвол.

Несколько примеров:

```
'abc' SIMILAR TO 'abc'      true
'abc' SIMILAR TO 'a'        false
'abc' SIMILAR TO '%(b|d)%'  true
'abc' SIMILAR TO '(b|c)%'   false
```

Функция `substring` с тремя параметрами, `substring(`*строка* `from` *шаблон* `for` *спецсимвол*`)` извлекает подстроку, соответствующую шаблону регулярного выражения SQL. Как и с `SIMILAR TO`, указанному шаблону должна соответствовать вся строка; в противном случае функция не найдёт ничего и вернёт NULL. Для обозначения части шаблона, которая должна быть возвращена в случае успеха, шаблон должен содержать два спецсимвола и кавычки (`"`) после каждого. Эта функция возвращает часть шаблона между двумя такими маркерами.

Несколько примеров с маркерами `#"`, выделяющими возвращаемую строку:

```
substring('foobar' from '%#"o_b#"%' for '#')   oob
substring('foobar' from '#"o_b#"%' for '#')    NULL
```

9.7.3. Регулярные выражения POSIX

В Таблице 9.14 перечислены все существующие операторы для проверки строк регулярными выражениями POSIX.

Таблица 9.14. Операторы регулярных выражений

Оператор	Описание	Пример
`~`	Проверяет соответствие регулярному выражению с учётом регистра	`'thomas' ~ '.*thomas.*'`
`~*`	Проверяет соответствие регулярному выражению без учёта регистра	`'thomas' ~* '.*Thomas.*'`
`!~`	Проверяет несоответствие регулярному выражению с учётом регистра	`'thomas' !~ '.*Thomas.*'`
`!~*`	Проверяет несоответствие регулярному выражению без учёта регистра	`'thomas' !~* '.*vadim.*'`

Регулярные выражения POSIX предоставляют более мощные средства поиска по шаблонам, чем операторы LIKE и SIMILAR TO. Во многих командах Unix, таких как egrep, sed и awk используется язык шаблонов, похожий на описанный здесь.

Регулярное выражение — это последовательность символов, представляющая собой краткое определение набора строк (*регулярное множество*). Строка считается соответствующей регулярному выражению, если она является членом регулярного множества, описываемого регулярным выражением. Как и для LIKE, символы шаблона непосредственно соответствуют символам строки, за исключением специальных символов языка регулярных выражений. При этом спецсимволы регулярных выражений отличается от спецсимволов LIKE. В отличие от шаблонов LIKE, регулярное выражение может совпадать с любой частью строки, если только оно не привязано явно к началу и/или концу строки.

Несколько примеров:

```
'abc' ~ 'abc'    true
'abc' ~ '^a'     true
'abc' ~ '(b|d)'  true
'abc' ~ '^(b|c)' false
```

Более подробно язык шаблонов в стиле POSIX описан ниже.

Функция substring с двумя параметрами, substring(*строка* from *шаблон*), извлекает подстроку, соответствующую шаблону регулярного выражения POSIX. Она возвращает фрагмент текста, подходящий шаблону, если таковой находится в строке, либо NULL в противном случае. Но если шаблон содержит скобки, она возвращает первое подвыражение, заключённое в скобки (то, которое начинается с самой первой открывающей скобки). Если вы хотите использовать скобки, но не в таком особом режиме, можно просто заключить в них всё выражение. Если же вам нужно включить скобки в шаблон до подвыражения, которое вы хотите извлечь, это можно сделать, используя группы без захвата, которые будут описаны ниже.

Несколько примеров:

```
substring('foobar' from 'o.b')     oob
substring('foobar' from 'o(.)b')   o
```

Функция regexp_replace подставляет другой текст вместо подстрок, соответствующих шаблонам регулярных выражений POSIX. Она имеет синтаксис regexp_replace(*исходная_строка*, *шаблон*, *замена* [, *флаги*]). Если *исходная_строка* не содержит фрагмента, подходящего под *шаблон*, она возвращается неизменной. Если же соответствие находится, возвращается *исходная_строка*, в которой вместо соответствующего фрагмента подставляется *замена*. Строка *замена* может содержать *n*, где *n* — число от 1 до 9, указывающее на исходный фрагмент, соответствующий *n*-ому подвыражению в скобках, и может содержать обозначение \&, указывающее, что будет вставлен фрагмент, соответствующий всему шаблону. Если же в текст замены нужно включить обратную косую черту буквально, следует написать \\. В необязательном параметре *флаги* передаётся текстовая строка, содержащая ноль или более однобуквенных флагов, меняющих поведение функции. Флаг i включает поиск без учёта регистра, а флаг g указывает, что заменяться должны все подходящие подстроки, а не только первая из них. Допустимые флаги (кроме g) описаны в Таблице 9.22.

Несколько примеров:

```
regexp_replace('foobarbaz', 'b..', 'X')
                                   fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g')
                                   fooXX
regexp_replace('foobarbaz', 'b(..)', 'X\1Y', 'g')
                                   fooXarYXazY
```

Функция regexp_match возвращает текстовый массив из всех подходящих подстрок, полученных из первого вхождения шаблона регулярного выражения POSIX в строке. Она имеет синтаксис

regexp_match(*строка*, *шаблон* [, *флаги*]). Если вхождение не находится, результатом будет NULL. Если вхождение находится и *шаблон* не содержит подвыражений в скобках, результатом будет текстовый массив с одним элементом, содержащим подстроку, соответствующую всему шаблону. Если вхождение находится и *шаблон* содержит подвыражения в скобках, результатом будет текстовый массив, в котором *n*-ым элементом будет *n*-ое заключённое в скобки подвыражение *шаблона* (не считая «незахватывающих» скобок; подробнее см. ниже). В параметре *флаги* передаётся необязательная текстовая строка, содержащая ноль или более однобуквенных флагов, меняющих поведение функции. Допустимые флаги описаны в Таблице 9.22.

Некоторые примеры:

```
SELECT regexp_match('foobarbequebaz', 'bar.*que');
 regexp_match
--------------
 {barbeque}
(1 row)

SELECT regexp_match('foobarbequebaz', '(bar)(beque)');
 regexp_match
--------------
 {bar,beque}
(1 row)
```

В общем случае просто получить всю найденную подстроку или NULL, если нет соответствия, можно примерно так:

```
SELECT (regexp_match('foobarbequebaz', 'bar.*que'))[1];
 regexp_match
--------------
 barbeque
(1 row)
```

Функция regexp_matches возвращает набор текстовых массивов со всеми подходящими подстроками, полученными в результате применения регулярного выражения POSIX к строке. Она имеет тот же синтаксис, что и regexp_match. Эта функция не возвращает никаких строк, если вхождений нет; возвращает одну строку, если найдено одно вхождение и не передан флаг g, или *N* строк, если найдено *N* вхождений и передан флаг g. Каждая возвращаемая строка представляет собой текстовый массив, содержащий всю найденную подстроку или подстроки, соответствующие заключённым в скобки подвыражениям *шаблона*, как и описанный выше результат regexp_match. Функция regexp_matches принимает все флаги, показанные в Таблице 9.22, а также флаг g, указывающий ей выдать все вхождения, а не только первое.

Несколько примеров:

```
SELECT regexp_matches('foo', 'not there');
 regexp_matches
----------------
(0 rows)

SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
 regexp_matches
----------------
 {bar,beque}
 {bazil,barf}
(2 rows)
```

Подсказка

В большинстве случаев regexp_matches() должна применяться с флагом g, так как если вас интересует только первое вхождение, проще и эффективнее использовать функцию

`regexp_match()`. Однако `regexp_match()` существует только в Postgres Pro версии 10 и выше. В старых версиях обычно помещали вызов `regexp_matches()` во вложенный SELECT, например, так:

```
SELECT col1, (SELECT regexp_matches(col2, '(bar)(beque)')) FROM tab;
```

В результате выдаётся текстовый массив, если вхождение найдено, или `NULL` в противном случае, так же как с `regexp_match()`. Без вложенного SELECT этот запрос не возвращает никакие строки, если соответствие не находится, а это обычно не то, что нужно.

Функция `regexp_split_to_table` разделяет строку, используя в качестве разделителя шаблон регулярного выражения POSIX. Она имеет синтаксис `regexp_split_to_table(`*`строка`*`, `*`шаблон`*` [, `*`флаги`*`])`. Если *`шаблон`* не находится в переданной строке, возвращается вся *`строка`* целиком. Если находится минимум одно вхождение, для каждого такого вхождения возвращается текст от конца предыдущего вхождения (или начала строки) до начала вхождения. После последнего найденного вхождения возвращается фрагмент от его конца до конца строки. В необязательном параметре *`флаги`* передаётся текстовая строка, содержащая ноль или более однобуквенных флагов, меняющих поведение функции. Флаги, которые поддерживает `regexp_split_to_table`, описаны в Таблице 9.22.

Функция `regexp_split_to_array` ведёт себя подобно `regexp_split_to_table`, за исключением того, что `regexp_split_to_array` возвращает результат в массиве элементов типа `text`. Она имеет синтаксис `regexp_split_to_array(`*`строка`*`, `*`шаблон`*` [, `*`флаги`*`])`. Параметры у этой функции те же, что и у `regexp_split_to_table`.

Несколько примеров:

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumps over the lazy dog',
 '\s+') AS foo;
  foo
-------
 the
 quick
 brown
 fox
 jumps
 over
 the
 lazy
 dog
(9 rows)

SELECT regexp_split_to_array('the quick brown fox jumps over the lazy dog', '\s+');
              regexp_split_to_array
-----------------------------------------------
 {the,quick,brown,fox,jumps,over,the,lazy,dog}
(1 row)

SELECT foo FROM regexp_split_to_table('the quick brown fox', '\s*') AS foo;
 foo
-----
 t
 h
 e
 q
 u
 i
 c
```

```
 k
 b
 r
 o
 w
 n
 f
 o
 x
(16 rows)
```

Как показывает последний пример, функции разделения по регулярным выражениям игнорируют вхождения нулевой длины, идущие в начале и в конце строки, а также непосредственно за предыдущим вхождением. Это поведение противоречит строгому определению поиска по регулярным выражениям, который реализуют функции `regexp_match` и `regexp_matches`, но обычно более удобно на практике. Подобное поведение наблюдается и в других программных средах, например в Perl.

9.7.3.1. Подробное описание регулярных выражений

Регулярные выражения в Postgres Pro реализованы с использованием программного пакета, который разработал Генри Спенсер (Henry Spencer). Практически всё следующее описание регулярных выражений дословно скопировано из его руководства.

Регулярное выражение (Regular expression, RE), согласно определению в POSIX 1003.2, может иметь две формы: *расширенное* RE или ERE (грубо говоря, это выражения, которые понимает `egrep`) и *простое* RE или BRE (грубо говоря, это выражения для `ed`). Postgres Pro поддерживает обе формы, а кроме того реализует некоторые расширения, не предусмотренные стандартом POSIX, но широко используемые вследствие их доступности в некоторых языках программирования, например в Perl и Tcl. Регулярные выражения, использующие эти несовместимые с POSIX расширения, здесь называются *усовершенствованными* RE или ARE. ARE практически представляют собой надмножество ERE, тогда как BRE отличаются некоторой несовместимостью в записи (помимо того, что они гораздо более ограничены). Сначала мы опишем формы ARE и ERE, отметив особенности, присущие только ARE, а затем расскажем, чем от них отличаются BRE.

> **Примечание**
>
> Postgres Pro изначально всегда предполагает, что регулярное выражение следует правилам ARE. Однако можно переключиться на более ограниченные правила ERE или BRE, добавив в шаблон RE *встроенный параметр*, как описано в Подразделе 9.7.3.4. Это может быть полезно для совместимости с приложениями, ожидающими от СУБД строгого следования правилам POSIX 1003.2.

Регулярное выражение определяется как одна или более *ветвей*, разделённых символами `|`. Оно считается соответствующим всему, что соответствует одной из этих ветвей.

Ветвь — это ноль или несколько *количественных атомов* или *ограничений*, соединённых вместе. Соответствие ветви в целом образуется из соответствия первой части, за которым следует соответствие второй части и т. д.; пустой ветви соответствует пустая строка.

Количественный атом — это *атом*, за которым может следовать *определитель количества*. Без этого определителя ему соответствует одно вхождение атома. С определителем количества ему может соответствовать некоторое число вхождений этого атома. Все возможные *атомы* перечислены в Таблице 9.15. Варианты определителей количества и их значения перечислены в Таблице 9.16.

Ограничению соответствует пустая строка, но это соответствие возможно только при выполнении определённых условий. Ограничения могут использоваться там же, где и атомы, за исключением

того, что их нельзя дополнять определителями количества. Простые ограничения показаны в Таблице 9.17; некоторые дополнительные ограничения описаны ниже.

Таблица 9.15. Атомы регулярных выражений

Атом	Описание
`( re)`	(где *re* — любое регулярное выражение) описывает соответствие *re*, при этом данное соответствие захватывается для последующей обработки
`(?: re)`	подобно предыдущему, но соответствие не захватывается (т. е. это набор скобок «без захвата») (применимо только к ARE)
`.`	соответствует любому символу
`[`*символы*`]`	*выражение в квадратных скобках*, соответствует любому из *символов* (подробнее это описано в Подразделе 9.7.3.2)
`\`*k*	(где *k* — не алфавитно-цифровой символ) соответствует обычному символу буквально, т. е. `\\` соответствует обратной косой черте
`\`*c*	где *c* — алфавитно-цифровой символ (за которым могут следовать другие символы), это *спецсимвол*, см. Подраздел 9.7.3.3 (применим только к ARE; в ERE и BRE этому атому соответствует *c*)
`{`	когда за этим символом следует любой символ, кроме цифры, этот атом соответствует левой фигурной скобке (`{`), если же за ним следует цифра, это обозначает начало *границы* (см. ниже)
x	(где *x* — один символ, не имеющий специального значения) соответствует этому символу

Выражение RE не может заканчиваться обратной косой чертой (`\`).

> **Примечание**
>
> Если параметр standard_conforming_strings выключен, каждый символ обратной косой черты, записываемый в текстовой константе, нужно дублировать. Подробнее это описано в Подразделе 4.1.2.1.

Таблица 9.16. Определители количества в регулярных выражениях

Определитель	Соответствует
`*`	0 или более вхождений атома
`+`	1 или более вхождений атома
`?`	0 или 1 вхождение атома
`{`*m*`}`	ровно *m* вхождений атома
`{`*m*`,}`	*m* или более вхождений атома
`{`*m*`,` *n*`}`	от *m* до *n* (включая границы) вхождений атома; *m* не может быть больше *n*

Определитель	Соответствует
*?	не жадная версия *
+?	не жадная версия +
??	не жадная версия ?
{m}?	не жадная версия {m}
{m,}?	не жадная версия {m,}
{m, n}?	не жадная версия {m, n}

В формах с {...} числа m и n определяют так называемые *границы* количества. Эти числа должны быть беззнаковыми десятичными целыми в диапазоне от 0 до 255 включительно.

Не жадные определители (допустимые только в ARE) описывают те же возможные соответствия, что и аналогичные им обычные («*жадные*»), но предпочитают выбирать наименьшее, а не наибольшее количество вхождений. Подробнее это описано в Подразделе 9.7.3.5.

> **Примечание**
>
> Определители количества не могут следовать один за другим, например запись ** будет ошибочной. Кроме того, определители не могут стоять в начале выражения или подвыражения и идти сразу после ^ или |.

Таблица 9.17. Ограничения в регулярных выражениях

Ограничение	Описание
^	соответствует началу строки
$	соответствует концу строки
(?= re)	*позитивный просмотр вперёд* находит соответствие там, где начинается подстрока, соответствующая re (только для ARE)
(?! re)	*негативный просмотр вперёд* находит соответствие там, где не начинается подстрока, соответствующая re (только для ARE)
(?<= re)	*позитивный просмотр назад* находит соответствие там, где заканчивается подстрока, соответствующая re (только для ARE)
(?<! re)	*негативный просмотр назад* находит соответствие там, где не заканчивается подстрока, соответствующая re (только для ARE)

Ограничения просмотра вперёд и назад не могут содержать *ссылки назад* (см. Подраздел 9.7.3.3), и все скобки в них считаются «скобками без захвата».

9.7.3.2. Выражения в квадратных скобках

Выражение в квадратных скобках содержит список символов, заключённый в []. Обычно ему соответствует любой символ из списка (об исключении написано ниже). Если список начинается с ^, ему соответствует любой символ, который *не* перечисляется далее в этом списке. Если два символа в списке разделяются знаком -, это воспринимается как краткая запись полного интервала символов между двумя заданными (и включая их) в порядке сортировки; например выражению [0-9] в ASCII соответствует любая десятичная цифра. Два интервала

не могут разделять одну границу, т. е. выражение `a-c-e` недопустимо. Интервалы зависят от порядка сортировки, который может меняться, поэтому в переносимых программах их лучше не использовать.

Чтобы включить в список `]`, этот символ нужно написать первым (сразу за `^`, если он присутствует). Чтобы включить в список символ `-`, его нужно написать первым или последним, либо как вторую границу интервала. Указать - в качестве первой границы интервал можно, заключив его между `[.` и `.]`, чтобы он стал элементом сортировки (см. ниже). За исключением этих символов, некоторых комбинаций с `[` (см. следующие абзацы) и спецсимволов (в ARE), все остальные специальные символы в квадратных скобках теряют своё особое значение. В частности, символ `\` по правилам ERE или BRE воспринимается как обычный, хотя в ARE он экранирует символ, следующий за ним.

Выражения в квадратных скобках могут содержать элемент сортировки (символ или последовательность символов или имя такой последовательности), определение которого заключается между `[.` и `.]`. Определяющая его последовательность воспринимается в выражении в скобках как один элемент. Это позволяет включать в такие выражения элементы, соответствующие последовательности нескольких символов. Например, с элементом сортировки `ch` в квадратных скобках регулярному выражению `[[.ch.]]*c` будут соответствовать первые пять символов строки `chchcc`.

> **Примечание**
>
> В настоящее время Postgres Pro не поддерживает элементы сортировки, состоящие из нескольких символов. Эта информация относится к возможному в будущем поведению.

В квадратных скобках могут содержаться элементы сортировки, заключённые между `[=` и `=]`, обозначающие *классы эквивалентности*, т. е. последовательности символов из всех элементов сортировки, эквивалентных указанному, включая его самого. (Если для этого символа нет эквивалентных, он обрабатывается как заключённый между `[.` и `.]`.) Например, если `e` и `ё` — члены одного класса эквивалентности, выражения `[[=e=]]`, `[[=ё=]]` и `[eё]` будут равнозначными. Класс эквивалентности нельзя указать в качестве границы интервала.

В квадратных скобках может также содержаться имя класса символов, заключённое между `[:` и `:]`, и заменяющее список всех символов этого класса. Стандартные имена классов: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper` и `xdigit`. Весь этот набор классов определён в ctype и он может меняться в зависимости от локали (языковой среды). Класс символов также нельзя использовать в качестве границы интервала.

Есть два особых вида выражений в квадратных скобках: выражения `[[:<:]]` и `[[:>:]]`, представляющие собой ограничения, соответствующие пустым строкам в начале и конце слова. Слово в данном контексте определяется как последовательность словосоставляющих символов, перед или после которой нет словосоставляющих символов. Словосоставляющий символ — это символ класса `alnum` (определённого в ctype) или подчёркивание. Это расширение совместимо со стандартом POSIX 1003.2, но не описано в нём, и поэтому его следует использовать с осторожностью там, где важна совместимость с другими системами. Обычно лучше использовать ограничивающие спецсимволы, описанные ниже; они также не совсем стандартны, но набрать их легче.

9.7.3.3. Спецсимволы регулярных выражений

Спецсимволы — это специальные команды, состоящие из `\` и последующего алфавитно-цифрового символа. Можно выделить следующие категории спецсимволов: обозначения символов, коды классов, ограничения и ссылки назад. Символ `\`, за которым идёт алфавитно-цифровой символ, не образующий допустимый спецсимвол, считается ошибочным в ARE. В ERE спецсимволов нет: вне квадратных скобок пара из `\` и последующего алфавитно-цифрового символа, воспринимается просто как данный символ, а в квадратных скобках и сам символ `\` воспринимается просто как обратная косая черта. (Последнее на самом деле нарушает совместимость между ERE и ARE.)

Спецобозначения символов введены для того, чтобы облегчить ввод в RE непечатаемых и других неудобных символов. Они приведены в Таблице 9.18.

Коды классов представляют собой краткий способ записи имён некоторых распространённых классов символов. Они перечислены в Таблице 9.19.

Спецсимволы ограничений обозначают ограничения, которым при совпадении определённых условий соответствует пустая строка. Они перечислены в Таблице 9.20.

Ссылка назад (`\n`) соответствует той же строке, какой соответствовало предыдущее подвыражение в скобках под номером *n* (см. Таблицу 9.21). Например, `([bc])\1` соответствует `bb` или `cc`, но не `bc` или `cb`. Это подвыражение должно полностью предшествовать ссылке назад в RE. Нумеруются подвыражения в порядке следования их открывающих скобок. При этом скобки без захвата исключаются из рассмотрения.

Таблица 9.18. Спецобозначения символов в регулярных выражениях

Спецсимвол	Описание
`\a`	символ звонка, как в C
`\b`	символ «забой», как в C
`\B`	синоним для обратной косой черты (`\`), сокращающий потребность в дублировании этого символа
`\c`*X*	(где *X* — любой символ) символ, младшие 5 бит которого те же, что и у *X*, а остальные равны 0
`\e`	символ, определённый в последовательности сортировки с именем `ESC`, либо, если таковой не определён, символ с восьмеричным значением `033`
`\f`	подача формы, как в C
`\n`	новая строка, как в C
`\r`	возврат каретки, как в C
`\t`	горизонтальная табуляция, как в C
`\u`*wxyz*	(где *wxyz* ровно четыре шестнадцатеричные цифры) символ с шестнадцатеричным кодом `0x`*wxyz*
`\U`*stuvwxyz*	(где *stuvwxyz* ровно восемь шестнадцатеричных цифр) символ с шестнадцатеричным кодом `0x`*stuvwxyz*
`\v`	вертикальная табуляция, как в C
`\x`*hhh*	(где *hhh* — несколько шестнадцатеричных цифр) символ с шестнадцатеричным кодом `0x`*hhh* (символ всегда один вне зависимости от числа шестнадцатеричных цифр)
`\0`	символ с кодом `0` (нулевой байт)
`\`*xy*	(где *xy* — ровно две восьмеричных цифры, не *ссылка назад*) символ с восьмеричным кодом `0`*xy*
`\`*xyz*	(где *xyz* — ровно три восьмеричных цифры, не *ссылка назад*) символ с восьмеричным кодом `0`*xyz*

Шестнадцатеричные цифры записываются символами `0-9` и `a-f` или `A-F`. Восьмеричные цифры — цифры от `0` до `7`.

Спецпоследовательности с числовыми кодами, задающими значения вне диапазона ASCII (0-127), воспринимаются по-разному в зависимости от кодировки базы данных. Когда база данных имеет кодировку UTF-8, спецкод равнозначен позиции символа в Unicode, например, `\u1234` обозначает символ `U+1234`. Для других многобайтных кодировок спецпоследовательности обычно просто задают серию байт, определяющих символ. Если в кодировке базы данных отсутствует символ, заданный спецпоследовательностью, ошибки не будет, но и никакие данные не будут ей соответствовать.

Символы, переданные спецобозначением, всегда воспринимаются как обычные символы. Например, `\135` кодирует `]` в ASCII, но спецпоследовательность `\135` не будет закрывать выражение в квадратных скобках.

Таблица 9.19. Спецкоды классов в регулярных выражениях

Спецсимвол	Описание
`\d`	`[[:digit:]]`
`\s`	`[[:space:]]`
`\w`	`[[:alnum:]_]` (подчёркивание также включается)
`\D`	`[^[:digit:]]`
`\S`	`[^[:space:]]`
`\W`	`[^[:alnum:]_]` (подчёркивание также включается)

В выражениях в квадратных скобках спецсимволы `\d`, `\s` и `\w` теряют свои внешние квадратные скобки, а `\D`, `\S` и `\W` — недопустимы. (Так что, например запись `[a-c\d]` равнозначна `[a-c[:digit:]]`. А запись `[a-c\D]`, которая была бы равнозначна `[a-c^[:digit:]]`, — недопустима.)

Таблица 9.20. Спецсимволы ограничений в регулярных выражений

Спецсимвол	Описание
`\A`	соответствует только началу строки (чем это отличается от `^`, описано в Подразделе 9.7.3.5)
`\m`	соответствует только началу слова
`\M`	соответствует только концу слова
`\y`	соответствует только началу или концу слова
`\Y`	соответствует только положению не в начале и не в конце слова
`\Z`	соответствует только концу строки (чем это отличается от `$`, описано в Подразделе 9.7.3.5)

Определением слова здесь служит то же, что было приведено выше в описании `[[:<:]]` и `[[:>:]]`. В квадратных скобках спецсимволы ограничений не допускаются.

Таблица 9.21. Ссылки назад в регулярных выражениях

Спецсимвол	Описание
`\`*`m`*	(где *`m`* — цифра, отличная от 0) — ссылка назад на подвыражение под номером *`m`*
`\`*`mnn`*	(где *`m`* — цифра, отличная от 0, а *`nn`* — ещё несколько цифр с десятичным значением *`mnn`*, не превышающим число закрытых до этого скобок

Спецсимвол	Описание
	с захватом) ссылка назад на подвыражение под номером *mnn*

> **Примечание**
>
> Регулярным выражениям присуща неоднозначность между восьмеричными кодами символов и ссылками назад, которая разрешается следующим образом (это упоминалось выше). Ведущий ноль всегда считается признаком восьмеричной последовательности. Единственная цифра, отличная от 0, за которой не следует ещё одна цифра, всегда воспринимается как ссылка назад. Последовательность из нескольких цифр, которая начинается не с 0, воспринимается как ссылка назад, если она идёт за подходящим подвыражением (т. е. число оказывается в диапазоне, допустимом для ссылки назад), в противном случае она воспринимается как восьмеричное число.

9.7.3.4. Метасинтаксис регулярных выражений

В дополнение к основному синтаксису, описанному выше, можно использовать также несколько особых форм и разнообразные синтаксические удобства.

Регулярное выражение может начинаться с одного из двух специальных префиксов режима. Если RE начинается с `***:`, его продолжение рассматривается как ARE. (В Postgres Pro это обычно не имеет значения, так как регулярные выражения воспринимаются как ARE по умолчанию; но это может быть полезно, когда параметр *флаги* функций regex включает режим ERE или BRE.) Если RE начинается с `***=`, его продолжение воспринимается как обычная текстовая строка, все его символы воспринимаются буквально.

ARE может начинаться со *встроенных параметров*: последовательности `(?xyz)` (где *xyz* — один или несколько алфавитно-цифровых символов), определяющих параметры остального регулярного выражения. Эти параметры переопределяют любые ранее определённые параметры, в частности они могут переопределить режим чувствительности к регистру, подразумеваемый для оператора regex, или параметр *флаги* функции regex. Допустимые буквы параметров показаны в Таблице 9.22. Заметьте, что те же буквы используются в параметре *флаги* функций regex.

Таблица 9.22. Буквы встроенных параметров ARE

Параметр	Описание
`b`	продолжение регулярного выражения — BRE
`c`	поиск соответствий с учётом регистра (переопределяет тип оператора)
`e`	продолжение RE — ERE
`i`	поиск соответствий без учёта регистра (см. Подраздел 9.7.3.5) (переопределяет тип оператора)
`m`	исторически сложившийся синоним `n`
`n`	поиск соответствий с учётом перевода строк (см. Подраздел 9.7.3.5)
`p`	переводы строк учитываются частично (см. Подраздел 9.7.3.5)
`q`	продолжение регулярного выражения — обычная строка («в кавычках»), содержимое которой воспринимается буквально
`s`	поиск соответствий без учёта перевода строк (по умолчанию)

Параметр	Описание
`t`	компактный синтаксис (по умолчанию; см. ниже)
`w`	переводы строк учитываются частично, но в другом, «странном» режиме (см. Подраздел 9.7.3.5)
`x`	развёрнутый синтаксис (см. ниже)

Внедрённые параметры начинают действовать сразу после скобки `)`, завершающей их последовательность. Они могут находиться только в начале ARE (после указания `***:`, если оно присутствует).

Помимо обычного (*компактного*) синтаксиса RE, в котором имеют значение все символы, поддерживается также *развёрнутый* синтаксис, включить который можно с помощью встроенного параметра `x`. В развёрнутом синтаксисе игнорируются пробельные символы, а также все символы от `#` до конца строки (или конца RE). Это позволяет разделять RE на строки и добавлять в него комментарии. Но есть три исключения:

- пробельный символ или `#`, за которым следует `\`, сохраняется
- пробельный символ или `#` внутри выражения в квадратных скобках сохраняется
- пробельные символы и комментарии не могут присутствовать в составных символах, например, в `(?:`

В данном контексте пробельными символами считаются пробел, табуляция, перевод строки и любой другой символ, относящийся к классу символов *space*.

И наконец, в ARE последовательность `(?#`*ttt*`)` (где *ttt* — любой текст, не содержащий `)`) вне квадратных скобок также считается комментарием и полностью игнорируется. При этом она так же не может находиться внутри составных символов, таких как `(?:`. Эти комментарии в большей степени историческое наследие, чем полезное средство; они считаются устаревшими, а вместо них рекомендуется использовать развёрнутый синтаксис.

Ни одно из этих расширений метасинтаксиса не будет работать, если выражение начинается с префикса `***=`, после которого строка воспринимается буквально, а не как RE.

9.7.3.5. Правила соответствия регулярным выражениям

В случае, когда RE может соответствовать более чем одной подстроке в заданной строке, соответствующей RE считается подстрока, которая начинается в ней первой. Если к данной позиции подобных соответствующих подстрок оказывается несколько, из них выбирается либо самая длинная, либо самая короткая из возможных, в зависимости от того, какой режим выбран в RE: *жадный* или *не жадный*.

Где жадный или не жадный характер RE определяется по следующим правилам:

- Большинство атомов и все ограничения не имеют признака жадности (так как они всё равно не могут соответствовать подстрокам разного состава).
- Скобки, окружающие RE, не влияют на его «жадность».
- Атом с определителем фиксированного количества (`{`*m*`}` или `{`*m*`}?`) имеет ту же характеристику жадности (или может не иметь её), как и сам атом.
- Атом с другими обычными определителями количества (включая `{`*m*`,`*n*`}`, где *m* равняется *n*) считается жадным (предпочитает соответствие максимальной длины).
- Атом с не жадным определителем количества (включая `{`*m*`,`*n*`}?`, где *m* равно *n*) считается не жадным (предпочитает соответствие минимальной длины).
- Ветвь (RE без оператора `|` на верхнем уровне) имеет ту же характеристику жадности, что и первый количественный атом в нём, имеющий атрибут жадности.

- RE, образованное из двух или более ветвей, соединённых оператором `|`, всегда считается жадным.

Эти правила связывают характеристики жадности не только с отдельными количественными атомами, но и с ветвями и целыми RE, содержащими количественные атомы. Это означает, что при сопоставлении ветвь или целое RE может соответствовать максимально длинной или короткой подстроке *в целом*. Когда определена длина всего соответствия, часть его, соответствующая конкретному подвыражению, определяется с учётом характеристики жадности для этого подвыражения, при этом подвыражения, начинающиеся в RE раньше, имеют больший приоритет, чем следующие за ними.

Это иллюстрирует следующий пример:

```
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
Результат: 123
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
Результат: 1
```

В первом случае RE в целом жадное, так как жадным является атом `Y*`. Соответствие ему начинается с буквы `Y` и оно включает подстроку максимальной длины с этого места, т. е. подстроку `Y123`. Результат выражения — её часть, соответствующая подвыражению в скобках, т. е. `123`. Во втором случае, RE в целом наследует не жадный характер от атома `Y*?`. Соответствие ему так же начинается с `Y`, но включает оно подстроку минимальной длины с этого места, т. е. `Y1`. И хотя подвыражение `[0-9]{1,3}` имеет жадный характер, оно не может повлиять на выбор длины соответствия в целом, поэтому ему остаётся только подстрока `1`.

Другими словами, когда RE содержит и жадные, и не жадные подвыражения, всё соответствие будет максимально длинным или коротким в зависимости от характеристики всего RE. Характеристики, связанные с подвыражениями, влияют только на то, какую часть подстроки может «поглотить» одно подвыражение относительно другого.

Чтобы явно придать характеристику «жадности» или «нежадности» подвыражению или всему RE, можно использовать определители `{1,1}` и `{1,1}?`, соответственно. Это полезно, когда вам нужно, чтобы общая характеристика жадности RE отличалась от той, что вытекает из его элементов. Например, предположим, что вы пытаетесь выделить из строки, содержащей несколько цифр, эти цифры и части до и после них. Можно попытаться сделать это так:

```
SELECT regexp_match('abc01234xyz', '(.*)(\d+)(.*)');
Результат: {abc0123,4,xyz}
```

Но это не будет работать: первая группа `.*` — «жадная», она «съест» всё, что сможет, оставляя для соответствия `\d+` только последнюю возможность, то есть последнюю цифру. Можно попытаться сделать запрос «нежадным»:

```
SELECT regexp_match('abc01234xyz', '(.*?)(\d+)(.*)');
Результат: {abc,0,""}
```

И это не будет работать, так теперь весь RE в целом стал нежадным, и все соответствия завершаются как можно раньше. Но мы можем получить нужный результат, явно сделав жадным всё RE:

```
SELECT regexp_match('abc01234xyz', '(?:(.*?)(\d+)(.*)){1,1}');
Результат: {abc,01234,xyz}
```

Управление общей характеристикой «жадности» RE независимо от «жадности» его компонентов даёт большую гибкость в описании шаблонов переменной длины.

При определении более длинного или более короткого соответствия длины соответствий определяются в символах, а не в элементах сортировки. Пустая строка считается длиннее, чем отсутствие соответствия. Например, выражению `bb*` соответствуют три символа в середине строки `abbbc`, выражению `(week|wee)(night|knights)` — все десять символов `weeknights`; когда выражение `(.*).*` сопоставляется со строкой `abc`, подвыражению в скобках соответствуют все три символа; а когда `(a*)*` сопоставляется со строкой `bc`, то и RE в целом, и подстроке в скобках соответствует пустая строка.

Игнорирование регистра символов даёт практически тот же эффект, как если бы в алфавите исчезли различия прописных и строчных букв. Если буква, существующая и в верхнем, и в нижнем регистре, фигурирует вне квадратных скобок как обычный символ, она по сути преобразуется в выражение в квадратных скобках, содержащее оба варианта, например `x` становится `[xX]`. Если же она фигурирует в выражении в квадратных скобках, в это выражение добавляются все её варианты, например `[x]` становится `[xX]`, а `[^x]` — `[^xX]`.

Когда включён режим учёта перевода строк, атом `.` и выражения в квадратных скобках с `^` никогда не будут соответствовать символам конца строки (так что соответствия никогда не будут пересекать границы строк, если в RE нет явных указаний на эти символы), а `^` и `$` будут соответствовать пустой подстроке не только в начале и конце всего текста, но и в начале и конце каждой отдельной его строки. Однако спецсимволы ARE `\A` и `\Z` по-прежнему будут соответствовать *только* началу и концу всего текста.

В режиме, когда переводы строк учитываются частично, особый смысл перевод строк имеет для атома `.` и выражений в квадратных скобках, но не для `^` и `$`.

В обратном частичном режиме, перевод строк имеет особый смысл для `^` и `$`, как и в режиме с учётом перевода строк, но не для `.` и выражений в квадратных скобках. Данный режим не очень полезен, но существует для симметрии.

9.7.3.6. Пределы и совместимость

В текущей реализации отсутствует какой-либо явно заданный предел длины RE. Однако, разрабатывая программы высокой степени переносимости, не следует применять RE длиннее 256 байт, так как другая POSIX-совместимая реализация может отказаться обрабатывать такие регулярные выражения.

Единственная особенность ARE, действительно несовместимая с ERE стандарта POSIX проявляется в том, что в ARE знак `\` не теряет своё специальное значение в квадратных скобках. Все другие расширения ARE используют синтаксические возможности, которые не определены, не допустимы или не поддерживаются в ERE; синтаксис переключения режимов (`***`) также выходит за рамки синтаксиса POSIX как для BRE, так и для ERE.

Многие расширения ARE заимствованы из языка Perl, но некоторые были изменены, оптимизированы, а несколько расширений Perl были исключены. В результате имеют место следующие несовместимости: атомы `\b` и `\B`, отсутствие специальной обработки завершающего перевода строки, добавление исключений в квадратных скобках в число случаев, когда учитывается перевод строк, особые условия для скобок и ссылок назад в ограничениях просмотра вперёд/назад и семантика «наиболее длинное/короткое соответствие» (вместо «первое соответствие»).

Важно отметить две несовместимости синтаксиса ARE и регулярных выражений ERE, которые воспринимал PostgreSQL до версии 7.4:

- В ARE `\` с последующим алфавитно-цифровым символом представляет либо спецсимвол, либо ошибочную последовательность, тогда как в предыдущих версиях так можно было записывать алфавитно-цифровые символы. Это не должно быть большой проблемой, так как раньше не было причин использовать такие последовательности.
- В ARE знак `\` сохраняет своё специальное значение в `[]`, поэтому, чтобы передать `\` в квадратных скобках буквально, его нужно записать как `\\`.

9.7.3.7. Простые регулярные выражения

BRE имеют ряд отличий от ERE. В BRE знаки `|`, `+` и `?` теряют специальное значение, а замены им нет. Границы количества окружаются символами `\{` и `\}`, тогда как `{` и `}` рассматриваются как обычные символы. Вложенные подвыражения помещаются между `\(` и `\)`, а `(` и `)` представляют обычные символы. Символ `^` воспринимается как обычный, если только он не находится в начале RE или подвыражения в скобках, `$` — тоже обычный символ, если он находится не в конце RE или в конце подвыражения в скобках, и `*` — обычный символ, когда он находится в начале RE или подвыражения в скобках (возможно, после начального `^`). И, наконец, в BRE работают ссылки

назад с одной цифрой, \< и \> — синонимы для [[:<:]] и [[:>:]], соответственно; никакие другие спецсимволы в BRE не поддерживаются.

9.8. Функции форматирования данных

Функции форматирования в Postgres Pro предоставляют богатый набор инструментов для преобразования самых разных типов данных (дата/время, целое, числа с плавающей и фиксированной точкой) в форматированные строки и обратно. Все они перечислены в Таблице 9.23. Все эти функции следует одному соглашению: в первом аргументе передаётся значение, которое нужно отформатировать, а во втором — шаблон, определяющий формат ввода или вывода.

Таблица 9.23. Функции форматирования

Функция	Тип результата	Описание	Пример
to_char(timestamp, text)	text	преобразует время в текст	to_char(current_timestamp, 'HH12:MI:SS')
to_char(interval, text)	text	преобразует интервал в текст	to_char(interval '15h 2m 12s', 'HH24:MI:SS')
to_char(int, text)	text	преобразует целое в текст	to_char(125, '999')
to_char (double precision, text)	text	преобразует плавающее одинарной/двойной точности в текст	to_char(125.8::real, '999D9')
to_char(numeric, text)	text	преобразует числовое значение в текст	to_char(-125.8, '999D99S')
to_date(text, text)	date	преобразует текст в дату	to_date('05 Dec 2000', 'DD Mon YYYY')
to_number(text, text)	numeric	преобразует текст в число	to_number('12, 454.8-', '99G999D9S')
to_timestamp(text, text)	timestamp with time zone	преобразует строку во время	to_timestamp('05 Dec 2000', 'DD Mon YYYY')

> **Примечание**
>
> Также имеется функция to_timestamp с одним аргументом; см. Таблицу 9.30.

> **Подсказка**
>
> Функции to_timestamp и to_date предназначены для работы с входными форматами, которые нельзя преобразовать простым приведением. Для большинства стандартных форматов даты/времени работает простое приведение исходной строки к требуемому типу и использовать его гораздо легче. Так же и функцию to_number нет необходимости использовать для стандартных представлений чисел.

Шаблон вывода to_char может содержать ряд кодов, которые распознаются при форматировании и заменяются соответствующими данными. Любой текст, который не является кодом, копируется

в результат в неизменном виде. Подобным образом, в строке шаблона ввода (для других функций) коды шаблона определяют, какие значения содержит передаваемая текстовая строка.

Все коды форматирования даты и времени перечислены в Таблице 9.24.

Таблица 9.24. Коды форматирования даты/времени

Код	Описание
HH	час (01-12)
HH12	час (01-12)
HH24	час (00-23)
MI	минута (00-59)
SS	секунда (00-59)
MS	миллисекунда (000-999)
US	микросекунда (000000-999999)
SSSS	число секунд с начала суток (0-86399)
AM, am, PM или pm	обозначение времени до/после полудня (без точек)
A.M., a.m., P.M. или p.m.	обозначение времени до/после полудня (с точками)
Y,YYY	год (4 или более цифр) с разделителем
YYYY	год (4 или более цифр)
YYY	последние 3 цифры года
YY	последние 2 цифры года
Y	последняя цифра года
IYYY	недельный год по ISO 8601 (4 или более цифр)
IYY	последние 3 цифры недельного года по ISO 8601
IY	последние 2 цифры недельного года по ISO 8601
I	последняя цифра недельного года по ISO 8601
BC, bc, AD или ad	обозначение эры (без точек)
B.C., b.c., A.D. или a.d.	обозначение эры (с точками)
MONTH	полное название месяца в верхнем регистре (дополненное пробелами до 9 символов)
Month	полное название месяца с большой буквы (дополненное пробелами до 9 символов)
month	полное название месяца в нижнем регистре (дополненное пробелами до 9 символов)
MON	сокращённое название месяца в верхнем регистре (3 буквы в английском; в других языках длина может меняться)
Mon	сокращённое название месяца с большой буквы (3 буквы в английском; в других языках длина может меняться)
mon	сокращённое название месяца в нижнем регистре (3 буквы в английском; в других языках длина может меняться)
MM	номер месяца (01-12)

Код	Описание
DAY	полное название дня недели в верхнем регистре (дополненное пробелами до 9 символов)
Day	полное название дня недели с большой буквы (дополненное пробелами до 9 символов)
day	полное название дня недели в нижнем регистре (дополненное пробелами до 9 символов)
DY	сокращённое название дня недели в верхнем регистре (3 буквы в английском; в других языках может меняться)
Dy	сокращённое название дня недели с большой буквы (3 буквы в английском; в других языках длина может меняться)
dy	сокращённое название дня недели в нижнем регистре (3 буквы в английском; в других языках длина может меняться)
DDD	номер дня в году (001-366)
IDDD	номер дня в году по ISO 8601 (001-371; 1 день — понедельник первой недели по ISO)
DD	день месяца (01-31)
D	номер дня недели, считая с воскресенья (1) до субботы (7)
ID	номер дня недели по ISO 8601, считая с понедельника (1) до воскресенья (7)
W	неделя месяца (1-5) (первая неделя начинается в первое число месяца)
WW	номер недели в году (1-53) (первая неделя начинается в первый день года)
IW	номер недели в году по ISO 8601 (01-53; первый четверг года относится к неделе 1)
CC	век (2 цифры) (двадцать первый век начался 2001-01-01)
J	юлианская дата (целое число дней от 24 ноября 4714 г. до н. э. 00:00 по местному времени; см. Раздел B.7)
Q	квартал
RM	номер месяца римскими цифрами в верхнем регистре (I-XII; I=январь)
rm	номер месяца римскими цифрами в нижнем регистре (i-xii; i=январь)
TZ	сокращённое название часового пояса в верхнем регистре (поддерживается только в to_char)
tz	сокращённое название часового пояса в нижнем регистре (поддерживается только в to_char)
OF	смещение часового пояса от UTC (поддерживается только в to_char)

К любым кодам форматирования можно добавить модификаторы, изменяющие их поведение. Например, шаблон форматирования FMMonth включает код Month с модификатором

FM. Модификаторы, предназначенные для форматирования даты/времени, перечислены в Таблице 9.25.

Таблица 9.25. Модификаторы кодов для форматирования даты/времени

Модификатор	Описание	Пример
Приставка FM	режим заполнения (подавляет ведущие нули и дополнение пробелами)	FMMonth
Окончание TH	окончание порядкового числительного в верхнем регистре	DDTH, например 12TH
Окончание th	окончание порядкового числительного в нижнем регистре	DDth, например 12th
Приставка FX	глобальный параметр фиксированного формата (см. замечания)	FX Month DD Day
Приставка TM	режим перевода (выводятся локализованные названия дней и месяцев, исходя из lc_time)	TMMonth
Окончание SP	режим числа прописью (не реализован)	DDSP

Замечания по использованию форматов даты/времени:

- FM подавляет дополняющие пробелы и нули справа, которые в противном случае будут добавлены, чтобы результат имел фиксированную ширину. В Postgres Pro модификатор FM действует только на следующий код, тогда как в Oracle FM её действие распространяется на все последующие коды, пока не будет отключено последующим модификатором FM.
- TM не затрагивает замыкающие пробелы. Функции to_timestamp и to_date игнорируют указание TM.
- to_timestamp и to_date пропускают повторяющиеся пробелы во входной строке, если только не используется параметр FX. Например, to_timestamp('2000 JUN', 'YYYY MON') будет работать, но to_timestamp('2000 JUN', 'FXYYYY MON') вернёт ошибку, так как to_timestamp в данном случае ожидает только один разделяющий пробел. Приставка FX должна быть первой в шаблоне.
- Шаблоны для функций to_char могут содержать обычный текст; он будет выведен в неизменном виде. Чтобы вывести текст принудительно, например, если в нём оказываются поддерживаемые коды, его можно заключить в кавычки. Например, в строке '"Hello Year "YYYY', код YYYY будет заменён номером года, а буква Y в слове Year останется неизменной. В функциях to_date, to_number и to_timestamp при обработке подстроки в кавычках просто пропускаются символы входной строки по числу символов в подстроке, например для "XX" будут пропущены два символа.
- Если вы хотите получить в результате кавычки, перед ними нужно добавить обратную косую черту, например так: '\"YYYY Month\"'.
- Если в функциях to_timestamp и to_date формат года определяется менее, чем 4 цифрами, например, как YYY, и в переданном значении года тоже меньше 4 цифр, год пересчитывается в максимально близкий к году 2020, т. е. 95 воспринимается как 1995.
- Функции to_timestamp и to_date воспринимают отрицательные значения годов как относящиеся к годам до н. э. Если же указать отрицательное значение и добавить явный

признак BC (до н. э.), год будет относиться к н. э. Нулевое значение года воспринимается как 1 год до н. э.

- В функциях to_timestamp и to_date с преобразованием YYYY связано ограничение, когда обрабатываемый год записывается более чем 4 цифрами. После YYYY необходимо будет добавить нецифровой символ или соответствующий код, иначе год всегда будет восприниматься как 4 цифры. Например, в to_date('200001131', 'YYYYMMDD') (с годом 20000) год будет интерпретирован как состоящий из 4 цифр; чтобы исправить ситуацию, нужно добавить нецифровой разделитель после года, как в to_date('20000-1131', 'YYYY-MMDD'), или код как в to_date('20000Nov31', 'YYYYMonDD').
- Функции to_timestamp и to_date принимают поле CC (век), но игнорируют его, если в шаблоне есть поле YYY, YYYY или Y,YYY. Если CC используется с YY или Y, результатом будет год в данном столетии. Если присутствует только код столетия, без года, подразумевается первый год этого века.
- Функции to_timestamp и to_date принимают названия и номера дней недели (DAY, D и связанные типы полей), но игнорируют их при вычислении результата. То же самое происходит с полями квартала (Q).
- Функциям to_timestamp и to_date можно передать даты по недельному календарю ISO 8601 (отличающиеся от григорианских) одним из двух способов:
  - Год, номер недели и дня недели: например, to_date('2006-42-4', 'IYYY-IW-ID') возвращает дату 2006-10-19. Если день недели опускается, он считается равным 1 (понедельнику).
  - Год и день года: например, to_date('2006-291', 'IYYY-IDDD') также возвращает 2006-10-19.

  Попытка ввести дату из смеси полей григорианского и недельного календаря ISO 8601 бессмысленна, поэтому это будет считаться ошибкой. В контексте ISO 8601 понятия «номер месяца» и «день месяца» не существуют, а в григорианском календаре нет понятия номера недели по ISO.

  Внимание

  Тогда как to_date не примет смесь полей григорианского и недельного календаря ISO, to_char способна на это, так как форматы вроде YYYY-MM-DD (IYYY-IDDD) могут быть полезны. Но избегайте форматов типа IYYY-MM-DD; в противном случае с датами в начале года возможны сюрпризы. (За дополнительными сведениями обратитесь к Подразделу 9.9.1.)

- Функция to_timestamp воспринимает поля миллисекунд (MS) или микросекунд (US) как дробную часть число секунд. Например, to_timestamp('12.3', 'SS.MS') — это не 3 миллисекунды, а 300, так как это значение воспринимается как 12 + 0.3 секунды. Это значит, что для формата SS.MS входные значения 12.3, 12.30 и 12.300 задают одно и то же число миллисекунд. Чтобы получить три миллисекунды, время нужно записать в виде 12.003, тогда оно будет воспринято как 12 + 0.003 = 12.003 сек.

  Ещё более сложный пример: to_timestamp('15:12:02.020.001230', 'HH24:MI:SS.MS.US') будет преобразовано в 15 часов, 12 минут и 2 секунды + 20 миллисекунд + 1230 микросекунд = 2.021230 seconds.
- Нумерация дней недели в to_char(..., 'ID') соответствует функции extract(isodow from ...), но нумерация to_char(..., 'D') не соответствует нумерации, принятой в extract(dow from ...).
- Функция to_char(interval) обрабатывает форматы HH и HH12 в рамках 12 часов, то есть 0 и 36 часов будут выводиться как 12, тогда как HH24 выводит число часов полностью, и для значений interval результат может превышать 23.

Коды форматирования числовых значений перечислены в Таблице 9.26.

Таблица 9.26. Коды форматирования чисел

Код	Описание
`9`	позиция цифры (может отсутствовать, если цифра незначащая)
`0`	позиция цифры (присутствует всегда, даже если цифра незначащая)
`.` (точка)	десятичная точка
`,` (запятая)	разделитель групп (тысяч)
`PR`	отрицательное значение в угловых скобках
`S`	знак, добавляемый к числу (с учётом локали)
`L`	символ денежной единицы (с учётом локали)
`D`	разделитель целой и дробной части числа (с учётом локали)
`G`	разделитель групп (с учётом локали)
`MI`	знак минус в заданной позиции (если число < 0)
`PL`	знак плюс в заданной позиции (если число > 0)
`SG`	знак плюс или минус в заданной позиции
`RN`	число римскими цифрами (в диапазоне от 1 до 3999)
`TH` или `th`	окончание порядкового числительного
`V`	сдвиг на заданное количество цифр (см. замечания)
`EEEE`	экспоненциальная запись числа

Замечания по использованию форматов чисел:

- `0` обозначает позицию цифры, которая будет выводиться всегда, даже если это незначащий ноль слева или справа. `9` также обозначает позицию цифры, но если это незначащий ноль слева, он заменяется пробелом, а если справа и задан режим заполнения, он удаляется. (Для функции `to_number()` эти два символа равнозначны.)
- Символы шаблона `S`, `L`, `D` и `G` представляют знак, символ денежной единицы, десятичную точку и разделитель тысяч, как их определяет текущая локаль (см. lc_monetary и lc_numeric). Символы точка и запятая представляют те же символы, обозначающие десятичную точку и разделитель тысяч, но не зависят от локали.
- Если в шаблоне `to_char()` отсутствует явное указание положения знака, для него резервируется одна позиция рядом с числом (слева от него). Если левее нескольких `9` помещён `S`, знак также будет приписан слева к числу.
- Знак числа, полученный кодами `SG`, `PL` или `MI`, не присоединяется к числу; например, `to_char(-12, 'MI9999')` выдаёт `'-  12'`, тогда как `to_char(-12, 'S9999')` — `'  -12'`. (В Oracle `MI` не может идти перед `9`, наоборот `9` нужно указать перед `MI`.)
- `TH` не преобразует значения меньше 0 и не поддерживает дробные числа.
- `PL`, `SG` и `TH` — расширения Postgres Pro.
- `V` с `to_char` умножает вводимое значение на `10^`*n*, где *n* — число цифр, следующих за `V`. `V` с `to_number` подобным образом делит значение. Функции `to_char` и `to_number` не поддерживают `V` с дробными числами (например, `99.9V99` не допускается).

- Код `EEEE` (научная запись) не может сочетаться с любыми другими вариантами форматирования или модификаторами, за исключением цифр и десятичной точки, и должен располагаться в конце строки шаблона (например, `9.99EEEE` — допустимый шаблон).

Для изменения поведения кодов к ним могут быть применены определённые модификаторы. Например, `FM99.99` обрабатывается как код `99.99` с модификатором `FM`. Все модификаторы для форматирования чисел перечислены в Таблице 9.27.

Таблица 9.27. Модификаторы шаблонов для форматирования чисел

Модификатор	Описание	Пример
Приставка `FM`	режим заполнения (подавляет завершающие нули и дополнение пробелами)	`FM99.99`
Окончание `TH`	окончание порядкового числительного в верхнем регистре	`999TH`
Окончание `th`	окончание порядкового числительного в нижнем регистре	`999th`

В Таблице 9.28 приведены некоторые примеры использования функции `to_char`.

Таблица 9.28. Примеры `to_char`

Выражение	Результат
`to_char(current_timestamp, 'Day, DD HH12:MI:SS')`	`'Tuesday , 06 05:39:18'`
`to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS')`	`'Tuesday, 6 05:39:18'`
`to_char(-0.1, '99.99')`	`' -.10'`
`to_char(-0.1, 'FM9.99')`	`'-.1'`
`to_char(-0.1, 'FM90.99')`	`'-0.1'`
`to_char(0.1, '0.9')`	`' 0.1'`
`to_char(12, '9990999.9')`	`' 0012.0'`
`to_char(12, 'FM9990999.9')`	`'0012.'`
`to_char(485, '999')`	`' 485'`
`to_char(-485, '999')`	`'-485'`
`to_char(485, '9 9 9')`	`' 4 8 5'`
`to_char(1485, '9,999')`	`' 1,485'`
`to_char(1485, '9G999')`	`' 1 485'`
`to_char(148.5, '999.999')`	`' 148.500'`
`to_char(148.5, 'FM999.999')`	`'148.5'`
`to_char(148.5, 'FM999.990')`	`'148.500'`
`to_char(148.5, '999D999')`	`' 148,500'`
`to_char(3148.5, '9G999D999')`	`' 3 148,500'`
`to_char(-485, '999S')`	`'485-'`
`to_char(-485, '999MI')`	`'485-'`
`to_char(485, '999MI')`	`'485 '`
`to_char(485, 'FM999MI')`	`'485'`

Выражение	Результат
to_char(485, 'PL999')	'+485'
to_char(485, 'SG999')	'+485'
to_char(-485, 'SG999')	'-485'
to_char(-485, '9SG99')	'4-85'
to_char(-485, '999PR')	'<485>'
to_char(485, 'L999')	'DM 485'
to_char(485, 'RN')	' CDLXXXV'
to_char(485, 'FMRN')	'CDLXXXV'
to_char(5.2, 'FMRN')	'V'
to_char(482, '999th')	' 482nd'
to_char(485, '"Good number:"999')	'Good number: 485'
to_char(485.8, '"Pre:"999" Post:" .999')	'Pre: 485 Post: .800'
to_char(12, '99V999')	' 12000'
to_char(12.4, '99V999')	' 12400'
to_char(12.45, '99V9')	' 125'
to_char(0.0004859, '9.99EEEE')	' 4.86e-04'

9.9. Операторы и функции даты/времени

Все существующие функции для обработки даты/времени перечислены в Таблице 9.30, а подробнее они описаны в следующих подразделах. Поведение основных арифметических операторов (+, * и т. д.) описано в Таблице 9.29. Функции форматирования этих типов данных были перечислены в Разделе 9.8. Общую информацию об этих типах вы получили (или можете получить) в Разделе 8.5.

Помимо этого, для типов даты/времени имеются обычные операторы сравнения, показанные в Таблице 9.1. Значения даты и даты со временем (с часовым поясом или без него) можно сравнивать как угодно, тогда как значения только времени (с часовым поясом или без него) и интервалы допустимо сравнивать, только если их типы совпадают. При сравнении даты со временем без часового пояса и даты со временем с часовым поясом предполагается, что первое значение задано в часовом поясе, установленном параметром TimeZone, и оно пересчитывается в UTC для сравнения со вторым значением (внутри уже представленным в UTC). Аналогичным образом, при сравнении значений даты и даты со времени первое считается соответствующим полночи в часовом поясе TimeZone.

Все описанные ниже функции и операторы принимают две разновидности типов time или timestamp: с часовым поясом (time with time zone и timestamp with time zone) и без него (time without time zone и timestamp without time zone). Для краткости здесь они рассматриваются вместе. Кроме того, операторы + и * обладают переместительным свойством (например, date + integer = integer + date); здесь будет приведён только один вариант для каждой пары.

Таблица 9.29. Операторы даты/времени

Оператор	Пример	Результат
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00:00'
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00:00'

Оператор	Пример	Результат
+	interval '1 day' + interval '1 hour'	interval '1 day 01:00:00'
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00:00'
+	time '01:00' + interval '3 hours'	time '04:00:00'
-	- interval '23 hours'	interval '-23:00:00'
-	date '2001-10-01' - date '2001-09-28'	integer '3' (дня)
-	date '2001-10-01' - integer '7'	date '2001-09-24'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00:00'
-	time '05:00' - time '03:00'	interval '02:00:00'
-	time '05:00' - interval '2 hours'	time '03:00:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28 00:00:00'
-	interval '1 day' - interval '1 hour'	interval '1 day -01:00:00'
-	timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'	interval '1 day 15:00:00'
*	900 * interval '1 second'	interval '00:15:00'
*	21 * interval '1 day'	interval '21 days'
*	double precision '3.5' * interval '1 hour'	interval '03:30:00'
/	interval '1 hour' / double precision '1.5'	interval '00:40:00'

Таблица 9.30. Функции даты/времени

Функция	Тип результата	Описание	Пример	Результат
age(timestamp, timestamp)	interval	Вычитает аргументы и выдаёт «символический» результат с годами и месяцами, а не просто днями	age(timestamp '2001-04-10', timestamp '1957-06-13')	43 years 9 mons 27 days (43 года 9 месяцев 27 дней)
age(timestamp)	interval	Вычитает дату/время из current_date (полночь текущего дня)	age(timestamp '1957-06-13')	43 years 8 mons 3 days (43 года 8 месяцев 3 дня)
clock_timestamp()	timestamp with time zone	Текущая дата и время (меняется в процессе выполнения		

Функция	Тип результата	Описание	Пример	Результат
		операторов); см. Подраздел 9.9.4		
current_date	date	Текущая дата; см. Подраздел 9.9.4		
current_time	time with time zone	Текущее время суток; см. Подраздел 9.9.4		
current_timestamp	timestamp with time zone	Текущая дата и время (на момент начала транзакции); см. Подраздел 9.9.4		
date_part(text, timestamp)	double precision	Возвращает поле даты (равнозначно extract); см. Подраздел 9.9.1	date_part('hour', timestamp '2001-02-16 20:38:40')	20
date_part(text, interval)	double precision	Возвращает поле даты (равнозначно extract); см. Подраздел 9.9.1	date_part('month', interval '2 years 3 months')	3
date_trunc(text, timestamp)	timestamp	Отсекает компоненты даты до заданной точности; см. также Подраздел 9.9.2	date_trunc('hour', timestamp '2001-02-16 20:38:40')	2001-02-16 20:00:00
date_trunc(text, interval)	interval	Отсекает компоненты даты до заданной точности; см. также Подраздел 9.9.2	date_trunc('hour', interval '2 days 3 hours 40 minutes')	2 days 03:00:00
extract(*field* from timestamp)	double precision	Возвращает поле даты; см. Подраздел 9.9.1	extract(hour from timestamp '2001-02-16 20:38:40')	20
extract(*field* from interval)	double precision	Возвращает поле даты; см. Подраздел 9.9.1	extract(month from interval '2 years 3 months')	3
isfinite(date)	boolean	Проверяет конечность даты (её отличие от +/- бесконечности)	isfinite(date '2001-02-16')	true
isfinite(timestamp)	boolean	Проверяет конечность времени (его отличие от +/- бесконечности)	isfinite(timestamp '2001-02-16 21:28:30')	true

Функция	Тип результата	Описание	Пример	Результат
isfinite(interval)	boolean	Проверяет конечность интервала	isfinite(interval '4 hours')	true
justify_days(interval)	interval	Преобразует интервал так, что каждый 30-дневный период считается одним месяцем	justify_days(interval '35 days')	1 mon 5 days (1 месяц 5 дней)
justify_hours(interval)	interval	Преобразует интервал так, что каждый 24-часовой период считается одним днём	justify_hours(interval '27 hours')	1 day 03:00:00 (1 день 03:00:00)
justify_ interval(interval)	interval	Преобразует интервал с применением justify_days и justify_hours и дополнительно корректирует знаки	justify_ interval(interval '1 mon -1 hour')	29 days 23:00:00 (29 дней 23:00:00)
localtime	time	Текущее время суток; см. Подраздел 9.9.4		
localtimestamp	timestamp	Текущая дата и время (на момент начала транзакции); см. Подраздел 9.9.4		
make_date(*year* int, *month* int, *day* int)	date	Образует дату из полей: year (год), month (месяц) и day (день)	make_date(2013, 7, 15)	2013-07-15
make_interval(*years* int DEFAULT 0, *months* int DEFAULT 0, *weeks* int DEFAULT 0, *days* int DEFAULT 0, *hours* int DEFAULT 0, *mins* int DEFAULT 0, *secs* double precision DEFAULT 0.0)	interval	Образует интервал из полей: years (годы), months (месяцы), weeks (недели), days (дни), hours (часы), minutes (минуты) и secs (секунды)	make_interval(days => 10)	10 days
make_time(*hour* int, *min* int, *sec* double precision)	time	Образует время из полей: hour (час), minute (минута) и sec (секунда)	make_time(8, 15, 23.5)	08:15:23.5

Функция	Тип результата	Описание	Пример	Результат
make_timestamp(*year* int, *month* int, *day* int, *hour* int, *min* int, *sec* double precision)	timestamp	Образует дату и время из полей: year (год), month (месяц), day (день), hour (час), minute (минута) и sec (секунда)	make_timestamp(2013, 7, 15, 8, 15, 23.5)	2013-07-15 08:15:23.5
make_timestamptz(*year* int, *month* int, *day* int, *hour* int, *min* int, *sec* double precision, [*timezone* text])	timestamp with time zone	Образует дату и время с часовым поясом из полей: year (год), month (месяц), day (день), hour (час), minute (минута) и sec (секунда). Если параметр *timezone* (часовой пояс) не указан, используется текущий часовой пояс.	make_timestamptz(2013, 7, 15, 8, 15, 23.5)	2013-07-15 08:15:23.5+01
now()	timestamp with time zone	Текущая дата и время (на момент начала транзакции); см. Подраздел 9.9.4		
statement_timestamp()	timestamp with time zone	Текущая дата и время (на момент начала текущего оператора); см. Подраздел 9.9.4		
timeofday()	text	Текущая дата и время (как clock_timestamp, но в виде строки типа text); см. Подраздел 9.9.4		
transaction_timestamp()	timestamp with time zone	Текущая дата и время (на момент начала транзакции); см. Подраздел 9.9.4		
to_timestamp(double precision)	timestamp with time zone	Преобразует время эпохи Unix (число секунд с 1970-01-01 00:00:00+00) в стандартное время	to_timestamp(1284352323)	2010-09-13 04:32:03+00

В дополнение к этим функциям поддерживается SQL-оператор OVERLAPS:

```
(начало1, конец1) OVERLAPS (начало2, конец2)
(начало1, длительность1) OVERLAPS (начало2, длительность2)
```

Его результатом будет true, когда два периода времени (определённые своими границами) пересекаются, и false в противном случае. Границы периода можно задать либо в виде пары дат, времени или дат со временем, либо как дату, время (или дату со временем) с интервалом. Когда указывается пара значений, первым может быть и начало, и конец периода: `OVERLAPS` автоматически считает началом периода меньшее значение. Периоды времени считаются наполовину открытыми, т. е. *начало<=время<конец*, если только *начало* и *конец* не равны — в этом случае период представляет один момент времени. Это означает, например, что два периода, имеющие только общую границу, не будут считаться пересекающимися.

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Результат:true
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Результат:false
SELECT (DATE '2001-10-29', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Результат:false
SELECT (DATE '2001-10-30', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Результат:true
```

При добавлении к значению типа `timestamp with time zone` значения `interval` (или при вычитании из него `interval`), поле дней в этой дате увеличивается (или уменьшается) на указанное число суток, а время суток остаётся неизменным. При пересечении границы перехода на летнее время (если в часовом поясе текущего сеанса производится этот переход) это означает, что `interval '1 day'` и `interval '24 hours'` не обязательно будут равны. Например, в часовом поясе `America/Denver`:

```
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval '1 day';
Результат: 2005-04-03 12:00:00-06
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval '24 hours';
Результат: 2005-04-03 13:00:00-06
```

Эта разница объясняется тем, что `2005-04-03 02:00` в часовом поясе `America/Denver` произошёл переход на летнее время.

Обратите внимание на возможную неоднозначность в поле `months` в результате функции `age`, вызванную тем, что число дней в разных месяцах неодинаково. Вычисляя оставшиеся дни месяца, Postgres Pro рассматривает месяц меньшей из двух дат. Например, результатом `age('2004-06-01', '2004-04-30')` будет `1 mon 1 day`, так как в апреле 30 дней, а то же выражение с датой 30 мая выдаст `1 mon 2 days`, так как в мае 31 день.

Вычитание дат и дат со временем также может быть нетривиальной операцией. Один принципиально простой способ выполнить такое вычисление — преобразовать каждое значение в количество секунд, используя `EXTRACT(EPOCH FROM ...)`, а затем найти разницу результатов; при этом будет получено число *секунд* между двумя датами. При этом будет учтено неодинаковое число дней в месяцах, изменения часовых поясов и переходы на летнее время. При вычитании дат или дат со временем с помощью оператора «-» выдаётся число дней (по 24 часа) и часов/минут/секунд между данными значениями, с учётом тех же факторов. Функция `age` возвращает число лет, месяцев, дней и часов/минут/секунд, выполняя вычитание по полям, а затем пересчитывая отрицательные значения. Различие этих подходов иллюстрируют следующие запросы. Показанные результаты были получены для часового пояса `'US/Eastern'`; между двумя заданными датами произошёл переход на летнее время:

```
SELECT EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
       EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00');
Результат:10537200
SELECT (EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
        EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00'))
```

```
        / 60 / 60 / 24;
Результат:121.958333333333
SELECT timestamptz '2013-07-01 12:00:00' - timestamptz '2013-03-01 12:00:00';
Результат:121 days 23:00:00
SELECT age(timestamptz '2013-07-01 12:00:00', timestamptz '2013-03-01 12:00:00');
Результат:4 mons
```

9.9.1. EXTRACT, date_part

```
EXTRACT(field FROM source)
```

Функция `extract` получает из значений даты/времени поля, такие как год или час. Здесь *источник* — значение типа `timestamp`, `time` или `interval`. (Выражения типа `date` приводятся к типу `timestamp`, так что допускается и этот тип.) Указанное *поле* представляет собой идентификатор, по которому из источника выбирается заданное поле. Функция `extract` возвращает значения типа `double precision`. Допустимые поля:

`century`

Век:

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
Результат:20
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:21
```

Первый век начался 0001-01-01 00:00:00, хотя люди в то время и не считали так. Это определение распространяется на все страны с григорианским календарём. Века с номером 0 не было; считается, что 1 наступил после -1. Если такое положение вещей вас не устраивает, направляйте жалобы по адресу: Ватикан, Собор Святого Петра, Папе.

`day`

Для значений `timestamp` это день месяца (1 - 31), для значений `interval` — число дней

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:16

SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
Результат:40
```

`decade`

Год, делённый на 10

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:200
```

`dow`

День недели, считая с воскресенья (`0`) до субботы (`6`)

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:5
```

Заметьте, что в `extract` дни недели нумеруются не так, как в функции `to_char(..., 'D')`.

`doy`

День года (1 - 365/366)

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:47
```

`epoch`

Для значений `timestamp with time zone` это число секунд с 1970-01-01 00:00:00 UTC (отрицательное для предшествующего времени); для значений `date` и `timestamp` —

номинальное число секунд с 1970-01-01 00:00:00 без учёта часового пояса, переходов на летнее время и т. п.; для значений `interval` — общее количество секунд в интервале

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40.12-08');
Результат:982384720.12

SELECT EXTRACT(EPOCH FROM TIMESTAMP '2001-02-16 20:38:40.12');
Результат:982355920.12

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Результат:442800
```

Преобразовать время эпохи назад, в значение `timestamp with time zone`, с помощью `to_timestamp` можно так:

```
SELECT to_timestamp(982384720.12);
Результат:2001-02-17 04:38:40.12+00
```

Имейте в виду, что применяя `to_timestamp` к времени эпохи, извлечённому из значения `date` или `timestamp`, можно получить не вполне ожидаемый результат: эта функция подразумевает, что изначальное значение задано в часовом поясе UTC, но это может быть не так.

`hour`

Час (0 - 23)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:20
```

`isodow`

День недели, считая с понедельника (`1`) до воскресенья (`7`)

```
SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
Результат:7
```

Результат отличается от `dow` только для воскресенья. Такая нумерация соответствует ISO 8601.

`isoyear`

Год по недельному календарю ISO 8601, в который попадает дата (неприменимо к интервалам)

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
Результат:2005
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
Результат:2006
```

Год по недельному календарю ISO начинается с понедельника недели, в которой оказывается 4 января, так что в начале января или в конце декабря год по ISO может отличаться от года по григорианскому календарю. Подробнее об этом рассказывается в описании поля `week`.

Этого поля не было в PostgreSQL до версии 8.3.

`julian`

Юлианская дата, соответствующая дате или дате/времени (для интервала не определена). Значение будет дробным, если заданное время отличается от начала суток по местному времени. За дополнительной информацией обратитесь к Разделу B.7.

```
SELECT EXTRACT(JULIAN FROM DATE '2006-01-01');
Результат:2453737
SELECT EXTRACT(JULIAN FROM TIMESTAMP '2006-01-01 12:00');
Результат:2453737.5
```

`microseconds`

Значение секунд с дробной частью, умноженное на 1 000 000; заметьте, что оно включает и целые секунды

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
Результат:28500000
```

`millennium`

Тысячелетие

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:3
```

Годы 20 века относятся ко второму тысячелетию. Третье тысячелетие началось 1 января 2001 г.

`milliseconds`

Значение секунд с дробной частью, умноженное на 1 000; заметьте, что оно включает и целые секунды.

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
Результат:28500
```

`minute`

Минуты (0 - 59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:38
```

`month`

Для значений `timestamp` это номер месяца в году (1 - 12), а для `interval` — остаток от деления числа месяцев на 12 (в интервале 0 - 11)

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:2

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
Результат:3

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
Результат:1
```

`quarter`

Квартал года (1 - 4), к которому относится дата

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:1
```

`second`

Секунды, включая дробную часть (0 - 59[1])

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:40

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
Результат:28.5
```

`timezone`

Смещение часового пояса от UTC, представленное в секундах. Положительные значения соответствуют часовым поясам к востоку от UTC, а отрицательные — к западу. (Строго говоря, в Postgres Pro используется не UTC, так как секунды координации не учитываются.)

`timezone_hour`

Поле часов в смещении часового пояса

[1] 60, если операционная система поддерживает секунды координации

`timezone_minute`

Поле минут в смещении часового пояса

`week`

Номер недели в году по недельному календарю ISO 8601. По определению, недели ISO 8601 начинаются с понедельника, а первая неделя года включает 4 января этого года. Другими словами, первый четверг года всегда оказывается в 1 неделе этого года.

В системе нумерации недель ISO первые числа января могут относиться к 52-ой или 53-ей неделе предыдущего года, а последние числа декабря — к первой неделе следующего года. Например, `2005-01-01` относится к 53-ей неделе 2004 г., а `2006-01-01` — к 52-ей неделе 2005 г., тогда как `2012-12-31` включается в первую неделю 2013 г. Поэтому для получения согласованных результатов рекомендуется использовать поле `isoyear` в паре с `week`.

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:7
```

`year`

Поле года. Учтите, что года `0` не было, и это следует иметь в виду, вычитая из годов нашей эры годы до нашей эры.

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:2001
```

> **Примечание**
>
> С аргументом +/-бесконечность `extract` возвращает +/-бесконечность для монотонно увеличивающихся полей (`epoch`, `julian`, `year`, `isoyear`, `decade`, `century` и `millennium`). Для других полей возвращается NULL. До версии 9.6 Postgres Pro возвращал ноль для всех случаев с бесконечными аргументами.

Функция `extract` в основном предназначена для вычислительных целей. Функции форматирования даты/времени описаны в Разделе 9.8.

Функция `date_part` эмулирует традиционный для Ingres эквивалент стандартной SQL-функции `extract`:

`date_part('`*поле*`', `*источник*`)`

Заметьте, что здесь параметр *поле* должен быть строковым значением, а не именем. Функция `date_part` воспринимает те же поля, что и `extract`.

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Результат:16

SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
Результат:4
```

9.9.2. date_trunc

Функция `date_trunc` работает подобно `trunc` для чисел.

`date_trunc('`*поле*`', `*значение*`)`

Здесь *значение* — это выражение типа `timestamp` или `interval`. (Значения типов `date` и `time` автоматически приводятся к типам `timestamp` и `interval`, соответственно.) Параметр *поле* определяет, до какой точности обрезать переданное значение. Возвращаемое значение будет иметь тип `timestamp` или `interval` и все его значения, менее значимые, чем заданное поле, будут равны нулю (или единице, если это номер дня или месяца).

Параметр *поле* может принимать следующие значения:

```
microseconds
milliseconds
second
minute
hour
day
week
month
quarter
year
decade
century
millennium
```

Примеры:

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
Результат: 2001-02-16 20:00:00

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Результат: 2001-01-01 00:00:00
```

9.9.3. AT TIME ZONE

Указание `AT TIME ZONE` позволяет переводить дату/время *без часового пояса* в дату/время *с часовым поясом* и обратно, а также пересчитывать значения *времени* для различных часовых поясов. Все разновидности этого указания проиллюстрированы в Таблице 9.31.

Таблица 9.31. Разновидности `AT TIME ZONE`

Выражение	Тип результата	Описание
`timestamp without time zone AT TIME ZONE` *часовой_пояс*	`timestamp with time zone`	Воспринимает заданное время *без указания часового пояса* как время в указанном часовом поясе
`timestamp with time zone AT TIME ZONE` *часовой_пояс*	`timestamp without time zone`	Переводит данное значение timestamp *с часовым поясом* в другой часовой пояс, но не сохраняет информацию о нём в результате
`time with time zone AT TIME ZONE` *часовой_пояс*	`time with time zone`	Переводит данное время *с часовым поясом* в другой часовой пояс

В этих выражениях желаемый *часовой_пояс* можно задать либо в виде текстовой строки (например, `'America/Los_Angeles'`), либо как интервал (например, `INTERVAL '-08:00'`). В первом случае название часового пояса можно указать любым из способов, описанных в Подразделе 8.5.3.

Примеры (в предположении, что местный часовой пояс `America/Los_Angeles`):

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'America/Denver';
Результат: 2001-02-16 19:38:40-08

SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'America/Denver';
Результат: 2001-02-16 18:38:40

SELECT TIMESTAMP '2001-02-16 20:38:40-05' AT TIME ZONE 'Asia/Tokyo' AT TIME ZONE
 'America/Chicago';
Результат: 2001-02-16 05:38:40
```

В первом примере для значения, заданного без часового пояса, указывается часовой пояс и полученное время выводится в текущем часовом поясе (заданном параметром TimeZone). Во втором примере значение времени смещается в заданный часовой пояс и выдаётся без указания часового пояса. Этот вариант позволяет хранить и выводить значения с часовым поясом, отличным от текущего. В третьем примере время в часовом поясе Токио пересчитывается для часового пояса Чикаго. При переводе значений *времени* без даты в другие часовые пояса используются определения часовых поясов, действующие в данный момент.

Функция timezone(*часовой_пояс*, *время*) равнозначна SQL-совместимой конструкции *время* AT TIME ZONE *часовой_пояс*.

9.9.4. Текущая дата/время

Postgres Pro предоставляет набор функций, результат которых зависит от текущей даты и времени. Все следующие функции соответствуют стандарту SQL и возвращают значения, отражающие время начала текущей транзакции:

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME(точность)
CURRENT_TIMESTAMP(точность)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME(точность)
LOCALTIMESTAMP(точность)
```

CURRENT_TIME и CURRENT_TIMESTAMP возвращают время с часовым поясом. В результатах LOCALTIME и LOCALTIMESTAMP нет информации о часовом поясе.

CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME и LOCALTIMESTAMP могут принимать необязательный параметр точности, определяющий, до какого знака после запятой следует округлять поле секунд. Если этот параметр отсутствует, результат будет иметь максимально возможную точность.

Несколько примеров:

```
SELECT CURRENT_TIME;
Результат: 14:39:53.662522-05

SELECT CURRENT_DATE;
Результат: 2001-12-23

SELECT CURRENT_TIMESTAMP;
Результат: 2001-12-23 14:39:53.662522-05

SELECT CURRENT_TIMESTAMP(2);
Результат: 2001-12-23 14:39:53.66-05

SELECT LOCALTIMESTAMP;
Результат: 2001-12-23 14:39:53.662522
```

Так как эти функции возвращают время начала текущей транзакции, во время транзакции эти значения не меняются. Это считается не ошибкой, а особенностью реализации: цель такого поведения в том, чтобы в одной транзакции «текущее» время было одинаковым и для разных изменений в одной транзакций записывалась одна отметка времени.

> **Примечание**
>
> В других СУБД эти значения могут изменяться чаще.

В Postgres Pro есть также функции, возвращающие время начала текущего оператора, а также текущее время в момент вызова функции. Таким образом, в Postgres Pro есть следующие функции, не описанные в стандарте SQL:

```
transaction_timestamp()
statement_timestamp()
clock_timestamp()
timeofday()
now()
```

Функция `transaction_timestamp()` равнозначна конструкции `CURRENT_TIMESTAMP`, но в её названии явно отражено, что она возвращает. Функция `statement_timestamp()` возвращает время начала текущего оператора (более точно, время получения последнего командного сообщения от клиента). Функции `statement_timestamp()` и `transaction_timestamp()` возвращают одно и то же значение в первой команде транзакции, но в последующих их показания будут расходиться. Функция `clock_timestamp()` возвращает фактическое текущее время, так что её значение меняется в рамках одной команды SQL. Функция `timeofday()` существует в Postgres Pro по историческим причинам и, подобно `clock_timestamp()`, она возвращает фактическое текущее время, но представленное в виде форматированной строки типа `text`, а не значения `timestamp with time zone`. Функция `now()` — традиционный для Postgres Pro эквивалент функции `transaction_timestamp()`.

Все типы даты/времени также принимают специальное буквальное значение `now`, подразумевающее текущую дату и время (тоже на момент начала транзакции). Таким образом, результат следующих трёх операторов будет одинаковым:

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';  -- см. замечание ниже
```

> **Подсказка**
>
> Не используйте третью форму для указания значения, которое будет вычисляться позднее, например, в предложении `DEFAULT` для столбца таблицы. Система преобразует `now` в значение `timestamp` в момент разбора константы, поэтому когда будет вставляться такое значение по умолчанию, в соответствующем столбце окажется время создания таблицы! Первые две формы будут вычисляться, только когда значение по умолчанию потребуется, так как это вызовы функции. Поэтому они дадут желаемый результат при добавлении строки в таблицу. (См. также Подраздел 8.5.1.4.)

9.9.5. Задержка выполнения

В случае необходимости вы можете приостановить выполнение серверного процесса, используя следующие функции:

```
pg_sleep(сек)
pg_sleep_for(interval)
pg_sleep_until(timestamp with time zone)
```

Функция `pg_sleep` переводит процесс текущего сеанса в спящее состояние на указанное число секунд (*сек*). Параметр *сек* имеет тип `double precision`, так что в нём можно указать и дробное число. Функция `pg_sleep_for` введена для удобства, ей можно передать большие значения задержки в типе `interval`. А `pg_sleep_until` удобнее использовать, когда необходимо задать определённое время выхода из спящего состояния. Например:

```
SELECT pg_sleep(1.5);
SELECT pg_sleep_for('5 minutes');
SELECT pg_sleep_until('tomorrow 03:00');
```

Примечание

Действительное разрешение интервала задержки зависит от платформы; обычно это 0.01. Фактическая длительность задержки не будет меньше указанного времени, но может быть больше, в зависимости, например от нагрузки на сервер. В частности, не гарантируется, что `pg_sleep_until` проснётся именно в указанное время, но она точно не проснётся раньше.

Предупреждение

Прежде чем вызывать `pg_sleep` или её вариации, убедитесь в том, что в текущем сеансе нет ненужных блокировок. В противном случае в состояние ожидания могут перейти и другие сеансы, так что это отразится на системе в целом.

9.10. Функции для перечислений

Для типов перечислений (описанных в Разделе 8.7) предусмотрено несколько функций, которые позволяют сделать код чище, не «зашивая» в нём конкретные значения перечисления. Эти функции перечислены в Таблице 9.32. В этих примерах подразумевается, что перечисление создано так:

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green',
 'blue', 'purple');
```

Таблица 9.32. Функции для перечислений

Функция	Описание	Пример	Результат примера
`enum_first(anyenum)`	Возвращает первое значение заданного перечисления	`enum_first( null::rainbow)`	`red`
`enum_last(anyenum)`	Возвращает последнее значение заданного перечисления	`enum_last( null::rainbow)`	`purple`
`enum_range(anyenum)`	Возвращает все значения заданного перечисления в упорядоченном массиве	`enum_range( null::rainbow)`	`{red,orange,yellow, green,blue,purple}`
`enum_range(anyenum, anyenum)`	Возвращает набор значений перечисления, лежащих между двумя заданными, в виде упорядоченного массива. Значения в параметрах должны принадлежать одному перечислению. Если в первом параметре передаётся NULL, результат функции начинается с первого значения перечисления, а если во втором —	`enum_range( 'orange'::rainbow, 'green'::rainbow)`	`{orange,yellow, green}`
		`enum_range(NULL, 'green'::rainbow)`	`{red,orange,yellow, green}`
		`enum_range( 'orange'::rainbow, NULL)`	`{orange,yellow, green,blue,purple}`

Функция	Описание	Пример	Результат примера
	заканчивается последним.		

Заметьте, что за исключением варианта `enum_range` с двумя аргументами, эти функции не обращают внимание на конкретное переданное им значение; их интересует только объявленный тип. Они возвращают один и тот же результат, когда им передаётся NULL или любое другое значение типа. Обычно эти функции применяются к столбцам таблицы или аргументам внешних функций, а не к предопределённым типам, как показано в этих примерах.

9.11. Геометрические функции и операторы

Для геометрических типов `point`, `box`, `lseg`, `line`, `path`, `polygon` и `circle` разработан большой набор встроенных функций и операторов, представленный в Таблице 9.33, Таблице 9.34 и Таблице 9.35.

> **Внимание**
>
> Заметьте, что оператор «идентичности», `~=`, представляет обычное сравнение на равенство значений `point`, `box`, `polygon` и `circle`. Для некоторых из этих типов определён также оператор `=`, но `=` проверяет только равенство *площадей*. Другие скалярные операторы сравнения (`<=` и т. д.) так же сравнивают площади значений этих типов.

Таблица 9.33. Геометрические операторы

Оператор	Описание	Пример
`+`	Сдвиг	`box '((0,0),(1,1))' + point '(2.0,0)'`
`-`	Сдвиг	`box '((0,0),(1,1))' - point '(2.0,0)'`
`*`	Масштабирование/поворот	`box '((0,0),(1,1))' * point '(2.0,0)'`
`/`	Масштабирование/поворот	`box '((0,0),(2,2))' / point '(2.0,0)'`
`#`	Точка или прямоугольник в пересечении	`box '((1,-1),(-1,1))' # box '((1,1),(-2,-2))'`
`#`	Число точек в пути или вершин в многоугольнике	`# path '((1,0),(0,1),(-1,0))'`
`@-@`	Длина, периметр или длина окружности	`@-@ path '((0,0),(1,0))'`
`@@`	Центр	`@@ circle '((0,0),10)'`
`##`	Точка, ближайшая к первому операнду и принадлежащая второму	`point '(0,0)' ## lseg '((2,0),(0,2))'`
`<->`	Расстояние между операндами	`circle '((0,0),1)' <-> circle '((5,0),1)'`
`&&`	Пересекаются ли операнды? (Для положительного ответа достаточно одной общей точки.)	`box '((0,0),(1,1))' && box '((0,0),(2,2))'`
`<<`	Строго слева?	`circle '((0,0),1)' << circle '((5,0),1)'`

Оператор	Описание	Пример
>>	Строго справа?	circle '((5,0),1)' >> circle '((0,0),1)'
&<	Не простирается правее?	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	Не простирается левее?	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<<|	Строго ниже?	box '((0,0),(3,3))' <<| box '((3,4),(5,5))'
|>>	Строго выше?	box '((3,4),(5,5))' |>> box '((0,0),(3,3))'
&<|	Не простирается выше?	box '((0,0),(1,1))' &<| box '((0,0),(2,2))'
|&>	Не простирается ниже?	box '((0,0),(3,3))' |&> box '((0,0),(2,2))'
<^	Ниже (может касаться)?	circle '((0,0),1)' <^ circle '((0,5),1)'
>^	Выше (может касаться)?	circle '((0,5),1)' >^ circle '((0,0),1)'
?#	Пересекает?	lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'
?-	Горизонтальный объект?	?- lseg '((-1,0),(1,0))'
?-	Выровнены по горизонтали?	point '(1,0)' ?- point '(0,0)'
?|	Вертикальный объект?	?| lseg '((-1,0),(1,0))'
?|	Выровнены по вертикали?	point '(0,1)' ?| point '(0,0)'
?-|	Перпендикулярны?	lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'
?||	Параллельны?	lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'
@>	Первый объект включает второй?	circle '((0,0),2)' @> point '(1,1)'
<@	Первый объект включён во второй?	point '(1,1)' <@ circle '((0,0),2)'
~=	Одинаковы?	polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'

Примечание

До PostgreSQL 8.2 операторы включения @> и <@ назывались соответственно ~ и @. Эти имена по-прежнему доступны, но считаются устаревшими и в конце концов будут удалены.

Таблица 9.34. Геометрические функции

Функция	Тип результата	Описание	Пример
area(*объект*)	double precision	площадь	area(box '((0,0), (1,1))')
center(*объект*)	point	центр	center(box '((0, 0),(1,2))')
diameter(circle)	double precision	диаметр круга	diameter(circle '((0,0),2.0)')
height(box)	double precision	вертикальный размер прямоугольника	height(box '((0, 0),(1,1))')
isclosed(path)	boolean	замкнутый путь?	isclosed(path '((0,0),(1,1),(2, 0))')
isopen(path)	boolean	открытый путь?	isopen(path '[(0, 0),(1,1),(2, 0)]')
length(*объект*)	double precision	длина	length(path '((-1, 0),(1,0))')
npoints(path)	int	число точек	npoints(path '[(0, 0),(1,1),(2, 0)]')
npoints(polygon)	int	число точек	npoints(polygon '((1,1),(0,0))')
pclose(path)	path	преобразует путь в замкнутый	pclose(path '[(0, 0),(1,1),(2, 0)]')
popen(path)	path	преобразует путь в открытый	popen(path '((0, 0),(1,1),(2, 0))')
radius(circle)	double precision	радиус окружности	radius(circle '((0,0),2.0)')
width(box)	double precision	горизонтальный размер прямоугольника	width(box '((0, 0),(1,1))')

Таблица 9.35. Функции преобразования геометрических типов

Функция	Тип результата	Описание	Пример
box(circle)	box	окружность в прямоугольник	box(circle '((0, 0),2.0)')
box(point)	box	точка в пустой прямоугольник	box(point '(0,0)')
box(point, point)	box	точки в прямоугольник	box(point '(0,0)', point '(1,1)')
box(polygon)	box	многоугольник в прямоугольник	box(polygon '((0, 0),(1,1),(2, 0))')
bound_box(box, box)	box	прямоугольники в окружающий прямоугольник	bound_box(box '((0,0),(1,1))',

Функция	Тип результата	Описание	Пример
			box '((3,3),(4,4))'
circle(box)	circle	прямоугольник в окружность	circle(box '((0,0),(1,1))')
circle(point, double precision)	circle	окружность из центра и радиуса	circle(point '(0,0)', 2.0)
circle(polygon)	circle	многоугольник в окружность	circle(polygon '((0,0),(1,1),(2,0))')
line(point, point)	line	точки в прямую	line(point '(-1,0)', point '(1,0)')
lseg(box)	lseg	диагональ прямоугольника в отрезок	lseg(box '((-1,0),(1,0))')
lseg(point, point)	lseg	точки в отрезок	lseg(point '(-1,0)', point '(1,0)')
path(polygon)	path	многоугольник в путь	path(polygon '((0,0),(1,1),(2,0))')
point(double precision, double precision)	point	образует точку	point(23.4, -44.5)
point(box)	point	центр прямоугольника	point(box '((-1,0),(1,0))')
point(circle)	point	центр окружности	point(circle '((0,0),2.0)')
point(lseg)	point	центр отрезка	point(lseg '((-1,0),(1,0))')
point(polygon)	point	центр многоугольника	point(polygon '((0,0),(1,1),(2,0))')
polygon(box)	polygon	прямоугольник в многоугольник с 4 вершинами	polygon(box '((0,0),(1,1))')
polygon(circle)	polygon	круг в многоугольник с 12 вершинами	polygon(circle '((0,0),2.0)')
polygon(*число_точек* , circle)	polygon	окружность с заданным *числом_точек*	polygon(12, circle '((0,0),2.0)')
polygon(path)	polygon	путь в многоугольник	polygon(path '((0,0),(1,1),(2,0))')

К двум компонентам типа `point` (точка) можно обратиться, как к элементам массива с индексами 0 и 1. Например, если `t.p` — столбец типа `point`, `SELECT p[0] FROM t` вернёт координату X, а `UPDATE t SET p[1] = ...` изменит координату Y. Таким же образом, значение типа `box` или `lseg` можно воспринимать как массив двух значений типа `point`.

Функция area работает с типами box, circle и path. При этом для типа path заданный путь не должен быть самопересекающимся. Например, эта функция не примет значение типа path '((0,0),(0,1),(2,1),(2,2),(1,2),(1,0),(0,0))'::PATH, но примет визуально идентичный путь '((0,0),(0,1),(1,1),(1,2),(2,2),(2,1),(1,1),(1,0),(0,0))'::PATH. Если вы не вполне поняли, что здесь подразумевается под самопересечением пути, нарисуйте на бумаге две фигуры по приведённым координатам.

9.12. Функции и операторы для работы с сетевыми адресами

В Таблица 9.36 показаны операторы, работающие с типами cidr и inet. Операторы <<, <<=, >>, >>= и && проверяют включения подсетей, рассматривая только биты сети в обоих адресах (игнорируя биты узлов) и определяя, идентична ли одна сеть другой или её подсети.

Таблица 9.36. Операторы для типов cidr и inet

Оператор	Описание	Пример
<	меньше	inet '192.168.1.5' < inet '192.168.1.6'
<=	меньше или равно	inet '192.168.1.5' <= inet '192.168.1.5'
=	равно	inet '192.168.1.5' = inet '192.168.1.5'
>=	больше или равно	inet '192.168.1.5' >= inet '192.168.1.5'
>	больше	inet '192.168.1.5' > inet '192.168.1.4'
<>	не равно	inet '192.168.1.5' <> inet '192.168.1.4'
<<	содержится в	inet '192.168.1.5' << inet '192.168.1/24'
<<=	равно или содержится в	inet '192.168.1/24' <<= inet '192.168.1/24'
>>	содержит	inet '192.168.1/24' >> inet '192.168.1.5'
>>=	равно или содержит	inet '192.168.1/24' >>= inet '192.168.1/24'
&&	содержит или содержится в	inet '192.168.1/24' && inet '192.168.1.80/28'
~	битовый NOT	~ inet '192.168.1.6'
&	битовый AND	inet '192.168.1.6' & inet '0.0.0.255'
|	битовый OR	inet '192.168.1.6' | inet '0.0.0.255'
+	сложение	inet '192.168.1.6' + 25
-	вычитание	inet '192.168.1.43' - 36
-	вычитание	inet '192.168.1.43' - inet '192.168.1.19'

В Таблице 9.37 перечислены функции, работающие с типами cidr и inet. Функции abbrev, host и text предназначены в основном для вывода данных в альтернативных форматах.

Таблица 9.37. Функции для типов `cidr` и `inet`

Функция	Тип результата	Описание	Пример	Результат
`abbrev( inet)`	`text`	вывод адрес в кратком текстовом виде	`abbrev(inet '10.1.0.0/16')`	`10.1.0.0/16`
`abbrev( cidr)`	`text`	вывод адрес в кратком текстовом виде	`abbrev(cidr '10.1.0.0/16')`	`10.1/16`
`broadcast( inet)`	`inet`	широковещательный адрес сети	`broadcast( '192.168.1.5/ 24')`	`192.168.1.255/ 24`
`family( inet)`	`int`	возвращает семейство адреса; `4` для адреса IPv4, `6` для IPv6	`family('::1')`	`6`
`host( inet)`	`text`	извлекает IP-адрес в виде текста	`host( '192.168.1.5/ 24')`	`192.168.1.5`
`hostmask( inet)`	`inet`	вычисляет маску узла для сетевого адреса	`hostmask( '192.168.23.20/ 30')`	`0.0.0.3`
`masklen( inet)`	`int`	выдаёт длину маски сети	`masklen( '192.168.1.5/ 24')`	`24`
`netmask( inet)`	`inet`	вычисляет маску сети для сетевого адреса	`netmask( '192.168.1.5/ 24')`	`255.255.255.0`
`network( inet)`	`cidr`	извлекает компонент сети из адреса	`network( '192.168.1.5/ 24')`	`192.168.1.0/24`
`set_masklen( inet, int)`	`inet`	задаёт размер маски для значения `inet`	`set_masklen( '192.168.1.5/ 24', 16)`	`192.168.1.5/16`
`set_masklen( cidr, int)`	`cidr`	задаёт размер маски для значения `cidr`	`set_masklen( '192.168.1.0/ 24'::cidr, 16)`	`192.168.0.0/16`
`text( inet)`	`text`	выводит в текстовом виде IP-адрес и длину маски	`text(inet '192.168.1.5')`	`192.168.1.5/32`
`inet_same_ family( inet, inet)`	`boolean`	адреса относятся к одному семейству?	`inet_same_ family( '192.168.1.5/24', '::1')`	`false`
`inet_merge( inet, inet)`	`cidr`	наименьшая сеть, включающая обе заданные сети	`inet_merge( '192.168.1.5/24', '192.168.2.5/24')`	`192.168.0.0/22`

Любое значение `cidr` можно привести к типу `inet`, явно или нет; поэтому все функции, показанные выше с типом `inet`, также будут работать со значениями `cidr`. (Некоторые из функций указаны отдельно для типов `inet` и `cidr`, потому что их поведение с разными типами различается.) Кроме

того, значение inet тоже можно привести к типу cidr. При этом все биты справа от сетевой маски просто обнуляются, чтобы значение стало допустимым для типа cidr. К типам inet и cidr можно привести и обычные текстовые значения, используя обычный синтаксис, например: inet(*выражение*) или *столбец*::cidr.

В Таблице 9.38 приведена функция, предназначенная для работы с типом macaddr. Функция trunc(macaddr) возвращает MAC-адрес, последние 3 байта в котором равны 0. Это может быть полезно для вычисления префикса, определяющего производителя.

Таблица 9.38. Функции macaddr

Функция	Тип результата	Описание	Пример	Результат
trunc(macaddr)	macaddr	обнуляет последние 3 байта	trunc(macaddr '12:34:56:78:90:ab')	12:34:56:00:00:00

Тип macaddr также поддерживает стандартные реляционные операторы лексической сортировки (>, <= и т. д.) и операторы битовой арифметики (~, & и |), соответствующие операциям NOT, AND и OR.

В Таблице 9.39 приведены функции, предназначенные для работы с типом macaddr8. Функция trunc(macaddr8) возвращает MAC-адрес, последние 5 байт в котором равны нулю. Это может быть полезно для вычисления префикса, определяющего производителя.

Таблица 9.39. Функции macaddr8

Функция	Тип результата	Описание	Пример	Результат
trunc(macaddr8)	macaddr8	обнуляет последние 5 байт	trunc(macaddr8 '12:34:56:78:90:ab:cd:ef')	12:34:56:00:00:00:00:0
macaddr8_ set7bit(macaddr8)	macaddr8	устанавливает в 7 бите единицу, чтобы получить так называемый модифицированный адрес EUI-64 (для включения в адрес IPv6)	macaddr8_ set7bit(macaddr8 '00:34:56:ab:cd:ef')	02:34:56:ff:fe:ab:cd:e

Тип macaddr8 также поддерживает стандартные реляционные операторы лексической сортировки (>, <= и т. д.) и операторы битовой арифметики (~, & и |), соответствующие операциям NOT, AND и OR.

9.13. Функции и операторы текстового поиска

В Таблице 9.40, Таблице 9.41 и Таблице 9.42 собраны все существующие функции и операторы, предназначенные для полнотекстового поиска. Во всех деталях возможности полнотекстового поиска в Postgres Pro описаны в Главе 12.

Таблица 9.40. Операторы текстового поиска

Оператор	Тип результата	Описание	Пример	Результат
@@	boolean	tsvector соответствует tsquery ?	to_tsvector('fat cats ate rats') @@ to_ tsquery('cat & rat')	t
@@@	boolean	устаревший синоним для @@	to_tsvector('fat cats ate	t

Оператор	Тип результата	Описание	Пример	Результат
			rats') @@@ to_tsquery('cat & rat')	
||	tsvector	объединяет два значения tsvector	'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector	'a':1 'b':2,5 'c':3 'd':4
&&	tsquery	логическое И (AND) двух запросов tsquery	'fat | rat'::tsquery && 'cat'::tsquery	('fat' | 'rat') & 'cat'
||	tsquery	логическое ИЛИ (OR) двух запросов tsquery	'fat | rat'::tsquery || 'cat'::tsquery	('fat' | 'rat') | 'cat'
!!	tsquery	отрицание запроса tsquery	!! 'cat'::tsquery	!'cat'
<->	tsquery	tsquery предшествует tsquery	to_tsquery('fat') <-> to_tsquery('rat')	'fat' <-> 'rat'
@>	boolean	запрос tsquery включает другой?	'cat'::tsquery @> 'cat & rat'::tsquery	f
<@	boolean	запрос tsquery включён в другой?	'cat'::tsquery <@ 'cat & rat'::tsquery	t

Примечание

Операторы включения tsquery рассматривают только лексемы двух запросов, игнорируя операторы их сочетания.

В дополнение к операторам, перечисленным в этой таблице, для типов tsvector и tsquery определены обычные операторы сравнения для B-дерева (=, < и т. д.). Они не очень полезны для поиска, но позволяют, в частности, создавать индексы для столбцов этих типов.

Таблица 9.41. Функции текстового поиска

Функция	Тип результата	Описание	Пример	Результат
array_to_tsvector(text[])	tsvector	преобразует массив лексем в tsvector	array_to_tsvector('{fat,cat, rat}'::text[])	'cat' 'fat' 'rat'
get_current_ts_config()	regconfig	получает конфигурацию текстового поиска по умолчанию	get_current_ts_config()	english
length(tsvector)	integer	число лексем в значении tsvector	length('fat:2, 4 cat:3 rat:5A'::tsvector)	3
numnode(tsquery)	integer	число лексем и операторов в запросе tsquery	numnode('(fat & rat) | cat'::tsquery)	5

Функция	Тип результата	Описание	Пример	Результат
plainto_tsquery([*конфигурация* regconfig ,] *запрос* text)	tsquery	выдаёт значение tsquery, игнорируя пунктуацию	plainto_tsquery('english', 'The Fat Rats')	'fat' & 'rat'
phraseto_tsquery([*конфигурация* regconfig ,] *запрос* text)	tsquery	выдаёт значение tsquery для поиска фразы, игнорируя пунктуацию	phraseto_tsquery('english', 'The Fat Rats')	'fat' <-> 'rat'
querytree(*запрос* tsquery)	text	получает индексируемую часть запроса tsquery	querytree('foo & ! bar'::tsquery)	'foo'
setweight(*вектор* tsvector, *вес* "char")	tsvector	назначает *вес* каждому элементу *вектора*	setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A')	'cat':3A 'fat':2A,4A 'rat':5A
setweight(*вектор* tsvector, *вес* "char", *лексемы* text[])	tsvector	назначает *вес* элементам *вектора*, перечисленным в массиве *лексемы*	setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A', '{cat, rat}')	'cat':3A 'fat':2, 4 'rat':5A
strip(tsvector)	tsvector	убирает позиции и веса из значения tsvector	strip('fat:2,4 cat:3 rat:5A'::tsvector)	'cat' 'fat' 'rat'
to_tsquery([*конфигурация* regconfig ,] *запрос* text)	tsquery	нормализует слова и переводит их в tsquery	to_tsquery('english', 'The & Fat & Rats')	'fat' & 'rat'
to_tsvector([*конфигурация* regconfig ,] *документ* text)	tsvector	сокращает текст документа до значения tsvector	to_tsvector('english', 'The Fat Rats')	'fat':2 'rat':3
to_tsvector([*конфигурация* regconfig ,] *документ* json(b))	tsvector	сокращает каждое строковое значение в документе до значения tsvector, а затем складывает эти значения по порядку в документе и выдаёт один tsvector	to_tsvector('english', '{"a": "The Fat Rats"}'::json)	'fat':2 'rat':3
ts_delete(*вектор* tsvector, *лексема* text)	tsvector	удаляет заданную *лексему* из *вектора*	ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, 'fat')	'cat':3 'rat':5A

Функция	**Тип результата**	**Описание**	**Пример**	**Результат**
ts_delete(*вектор* tsvector, *лексемы* text[])	tsvector	удаляет все вхождения лексем, перечисленных в массиве *лексемы*, из *вектора*	ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, ARRAY['fat', 'rat'])	'cat':3
ts_filter(*вектор* tsvector, *веса* "char"[])	tsvector	выбирает из *вектора* только элементы с заданным *весом*	ts_filter('fat:2,4 cat:3b rat:5A'::tsvector, '{a,b}')	'cat':3B 'rat':5A
ts_headline([*конфигурация* regconfig,] *документ* text, *запрос* tsquery [, *параметры* text])	text	выводит фрагмент, соответствующий запросу	ts_headline('x y z', 'z'::tsquery)	x y z
ts_headline([*конфигурация* regconfig,] *документ* json(b) , *запрос* tsquery [, *параметры* text])	text	выводит фрагмент, соответствующий запросу	ts_headline('{"a":"x y z"}'::json, 'z'::tsquery)	{"a":"x y z"}
ts_rank([*веса* float4[],] *вектор* tsvector, *запрос* tsquery [, *нормализация* integer])	float4	вычисляет ранг документа по отношению к запросу	ts_rank(textsearch, query)	0.818
ts_rank_cd([*веса* float4[],] *вектор* tsvector, *запрос* tsquery [, *нормализация* integer])	float4	вычисляет ранг документа по отношению к запросу, используя плотность покрытия (CDR)	ts_rank_cd('{0.1, 0.2, 0.4, 1.0}', textsearch, query)	2.01317
ts_rewrite(*запрос* tsquery, *цель* tsquery, *замена* tsquery)	tsquery	подставляет в запросе вместо *цели замену*	ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo| bar'::tsquery)	'b' & ('foo' | 'bar')
ts_rewrite(*запрос* tsquery, *выборка* text)	tsquery	заменяет элементы запроса, выбирая цели и подстановки командой SELECT	SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases')	'b' & ('foo' | 'bar')
tsquery_phrase(*запрос1* tsquery,	tsquery	создаёт запрос, который ищет *запрос1*, за которым идёт	tsquery_phrase(to_tsquery('fat'), to_	'fat' <-> 'cat'

Функция	Тип результата	Описание	Пример	Результат
запрос2 tsquery)		*запрос2* (как делает оператор <->)	tsquery('cat'))	
tsquery_phrase(*запрос1* tsquery, *запрос2* tsquery, *расстояние* integer)	tsquery	создаёт запрос, который ищет *запрос1*, за которым идёт *запрос2* на заданном *расстоянии*	tsquery_phrase(to_tsquery('fat'), to_ tsquery('cat'), 10)	'fat' <10> 'cat'
tsvector_to_ array(tsvector)	text[]	преобразует tsvector в массив лексем	tsvector_to_ array('fat:2, 4 cat:3 rat:5A'::tsvector)	{cat,fat,rat}
tsvector_ update_ trigger()	trigger	триггерная функция для автоматического изменения столбца типа tsvector	CREATE TRIGGER ... tsvector_ update_ trigger(tsvcol, 'pg_ catalog. swedish', title, body)	
tsvector_ update_ trigger_ column()	trigger	триггерная функция для автоматического изменения столбца типа tsvector	CREATE TRIGGER ... tsvector_ update_ trigger_ column(tsvcol, configcol, title, body)	
unnest(tsvector, OUT *лексема* text, OUT *позиции* smallint[], OUT *веса* text)	setof record	разворачивает tsvector в набор строк	unnest('fat:2, 4 cat:3 rat:5A'::tsvector)	(cat,{3}, {D}) ...

Примечание

Все функции текстового поиска, принимающие необязательный аргумент regconfig, будут использовать конфигурацию, указанную в параметре default_text_search_config, когда этот аргумент опущен.

Функции в Таблице 9.42 перечислены отдельно, так как они не очень полезны в традиционных операциях поиска. Они предназначены в основном для разработки и отладки новых конфигураций текстового поиска.

Таблица 9.42. Функции отладки текстового поиска

Функция	Тип результата	Описание	Пример	Результат
ts_debug([*конфигурация*	setof record	проверяет конфигурацию	ts_debug('english', 'The	(asciiword, "Word, all

Функция	Тип результата	Описание	Пример	Результат
regconfig,] *документ* text, OUT *псевдоним* text, OUT *описание* text, OUT *фрагмент* text, OUT *словари* regdictionary[], OUT *словарь* regdictionary, OUT *лексемы* text[])			Brightest supernovaes')	ASCII",The,{ english_stem },english_ stem,{}) ...
ts_lexize(*словарь* regdictionary, *фрагмент* text)	text[]	проверяет словарь	ts_lexize('english_ stem', 'stars')	{star}
ts_parse(*имя_ анализатора* text, *документ* text, OUT *код_ фрагмента* integer, OUT *фрагмент* text)	setof record	проверяет анализатор	ts_parse('default', 'foo - bar')	(1,foo) ...
ts_parse(*oid_ анализатора* oid, *документ* text, OUT *код_ фрагмента* integer, OUT *фрагмент* text)	setof record	проверяет анализатор	ts_parse(3722, 'foo - bar')	(1,foo) ...
ts_token_type(*имя_ анализатора* text, OUT *код_ фрагмента* integer, OUT *псевдоним* text, OUT *описание* text)	setof record	получает типы фрагментов, определённые анализатором	ts_token_type('default')	(1,asciiword, "Word, all ASCII") ...
ts_token_type(*oid_ анализатора* oid, OUT *код_ фрагмента* integer, OUT *псевдоним* text, OUT *описание* text)	setof record	получает типы фрагментов, определённые анализатором	ts_token_type(3722)	(1,asciiword, "Word, all ASCII") ...
ts_stat(*sql_ запрос* text, [*веса* text,]	setof record	получает статистику столбца tsvector	ts_stat('SELECT vector from apod')	(foo,10, 15) ...

Функция	Тип результата	Описание	Пример	Результат
OUT *слово* text, OUT *число_док* integer, OUT *число_вхожд* integer)				

9.14. XML-функции

Функции и подобные им выражения, описанные в этом разделе, работают со значениями типа `xml`. Информацию о типе `xml` вы можете найти в Разделе 8.13. Подобные функциям выражения `xmlparse` и `xmlserialize`, преобразующие значения `xml` в текст и обратно, здесь повторно не рассматриваются.

Для использования этих функций Postgres Pro нужно задействовать соответствующую библиотеку при сборке: `configure --with-libxml`.

9.14.1. Создание XML-контента

Для получения XML-контента из данных SQL существует целый набор функций и функциональных выражений, особенно полезных для выдачи клиентским приложениям результатов запроса в виде XML-документов.

9.14.1.1. xmlcomment

```
xmlcomment(текст)
```

Функция `xmlcomment` создаёт XML-значение, содержащее XML-комментарий с заданным текстом. Этот текст не должен содержать «--» или заканчиваться знаком «-», чтобы результирующая конструкция была допустимой в XML. Если аргумент этой функции NULL, результатом её тоже будет NULL.

Пример:

```
SELECT xmlcomment('hello');

  xmlcomment
--------------
 <!--hello-->
```

9.14.1.2. xmlconcat

```
xmlconcat(xml[, ...])
```

Функция `xmlconcat` объединяет несколько XML-значений и выдаёт в результате один фрагмент XML-контента. Значения NULL отбрасываются, так что результат будет равен NULL, только если все аргументы равны NULL.

Пример:

```
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');

      xmlconcat
----------------------
 <abc/><bar>foo</bar>
```

XML-объявления, если они присутствуют, обрабатываются следующим образом. Если во всех аргументах содержатся объявления одной версии XML, эта версия будет выдана в результате; в противном случае версии не будет. Если во всех аргументах определён атрибут standalone со значением «yes», это же значение будет выдано в результате. Если во всех аргументах

есть объявление standalone, но минимум в одном со значением «no», в результате будет это значение. В противном случае в результате не будет объявления standalone. Если же окажется, что в результате должно присутствовать объявление standalone, а версия не определена, тогда в результате будет выведена версия 1.0, так как XML-объявление не будет допустимым без указания версии. Указания кодировки игнорируются и будут удалены в любых случаях.

Пример:

```
SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml version="1.1" standalone="no"?
><bar/>');

             xmlconcat
-----------------------------------
 <?xml version="1.1"?><foo/><bar/>
```

9.14.1.3. xmlelement

```
xmlelement(name имя [, xmlattributes(значение [AS атрибут] [, ...])]
  [, содержимое, ...])
```

Выражение `xmlelement` создаёт XML-элемент с заданным именем, атрибутами и содержимым.

Примеры:

```
SELECT xmlelement(name foo);

 xmlelement
------------
 <foo/>

SELECT xmlelement(name foo, xmlattributes('xyz' as bar));

    xmlelement
------------------
 <foo bar="xyz"/>

SELECT xmlelement(name foo, xmlattributes(current_date as bar), 'cont', 'ent');

             xmlelement
-------------------------------------
 <foo bar="2007-01-26">content</foo>
```

Если имена элементов и атрибутов содержат символы, недопустимые в XML, эти символы заменяются последовательностями *_xHHHH_*, где *HHHH* — шестнадцатеричный код символа в Unicode. Например:

```
SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));

            xmlelement
----------------------------------
 <foo_x0024_bar a_x0026_b="xyz"/>
```

Если в качестве значения атрибута используется столбец таблицы, имя атрибута можно не указывать явно, этим именем станет имя столбца. Во всех остальных случаях имя атрибута должно быть определено явно. Таким образом, это выражение допустимо:

```
CREATE TABLE test (a xml, b xml);
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;
```

А следующие варианты — нет:

```
SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM test;
```

```
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;
```

Содержимое элемента, если оно задано, будет форматировано согласно его типу данных. Когда оно само имеет тип `xml`, из него можно конструировать сложные XML-документы. Например:

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),
                            xmlelement(name abc),
                            xmlcomment('test'),
                            xmlelement(name xyz));

                  xmlelement
----------------------------------------------
 <foo bar="xyz"><abc/><!--test--><xyz/></foo>
```

Содержимое других типов будет оформлено в виде допустимых символьных данных XML. Это, в частности, означает, что символы <, > и & будут преобразованы в сущности XML. Двоичные данные (данные типа `bytea`) представляются в кодировке base64 или в шестнадцатеричном виде, в зависимости от значения параметра xmlbinary. Следует ожидать, что конкретные представления отдельных типов данных могут быть изменены для приведения преобразований Postgres Pro в соответствие со стандартом SQL:2006 и новее, как описано в Подразделе D.3.1.3.

9.14.1.4. xmlforest

```
xmlforest(содержимое [AS имя] [, ...])
```

Выражение `xmlforest` создаёт последовательность XML-элементов с заданными именами и содержимым.

Примеры:

```
SELECT xmlforest('abc' AS foo, 123 AS bar);

          xmlforest
------------------------------
 <foo>abc</foo><bar>123</bar>


SELECT xmlforest(table_name, column_name)
FROM information_schema.columns
WHERE table_schema = 'pg_catalog';

                                       xmlforest
-----------------------------------------------------------------------------------------
 <table_name>pg_authid</table_name><column_name>rolname</column_name>
 <table_name>pg_authid</table_name><column_name>rolsuper</column_name>
 ...
```

Как показано во втором примере, имя элемента можно опустить, если источником содержимого служит столбец (в этом случае именем элемента по умолчанию будет имя столбца). В противном случае это имя необходимо указывать.

Имена элементов с символами, недопустимыми для XML, преобразуются так же, как и для `xmlelement`. Данные содержимого тоже приводятся к виду, допустимому для XML (кроме данных, которые уже имеют тип `xml`).

Заметьте, что такие XML-последовательности не являются допустимыми XML-документами, если они содержат больше одного элемента на верхнем уровне, поэтому может иметь смысл вложить выражения `xmlforest` в `xmlelement`.

9.14.1.5. xmlpi

```
xmlpi(name цель [, содержимое])
```

Выражение xmlpi создаёт инструкцию обработки XML. Содержимое, если оно задано, не должно содержать последовательность символов ?>.

Пример:

```
SELECT xmlpi(name php, 'echo "hello world";');

            xmlpi
-----------------------------
 <?php echo "hello world";?>
```

9.14.1.6. xmlroot

```
xmlroot(xml, version текст | нет значения [, standalone yes|no|нет значения])
```

Выражение xmlroot изменяет свойства корневого узла XML-значения. Если в нём указывается версия, она заменяет значение в объявлении версии корневого узла; также в корневой узел переносится значение свойства standalone.

```
SELECT xmlroot(xmlparse(document '<?xml version="1.1"?><content>abc</content>'),
               version '1.0', standalone yes);

                xmlroot
----------------------------------------
 <?xml version="1.0" standalone="yes"?>
 <content>abc</content>
```

9.14.1.7. xmlagg

```
xmlagg(xml)
```

Функция xmlagg, в отличие от других описанных здесь функций, является агрегатной. Она соединяет значения, поступающие на вход агрегатной функции, подобно функции xmlconcat, но делает это, обрабатывая множество строк, а не несколько выражений в одной строке. Дополнительно агрегатные функции описаны в Разделе 9.20.

Пример:

```
CREATE TABLE test (y int, x xml);
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');
SELECT xmlagg(x) FROM test;
        xmlagg
----------------------
 <foo>abc</foo><bar/>
```

Чтобы задать порядок сложения элементов, в агрегатный вызов можно добавить предложение ORDER BY, описанное в Подразделе 4.2.7. Например:

```
SELECT xmlagg(x ORDER BY y DESC) FROM test;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

Следующий нестандартный подход рекомендовался в предыдущих версиях и может быть по-прежнему полезен в некоторых случаях:

```
SELECT xmlagg(x) FROM (SELECT * FROM test ORDER BY y DESC) AS tab;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

9.14.2. Условия с XML

Описанные в этом разделе выражения проверяют свойства значений `xml`.

9.14.2.1. `IS DOCUMENT`

```
xml IS DOCUMENT
```

Выражение `IS DOCUMENT` возвращает true, если аргумент представляет собой правильный XML-документ, false в противном случае (т. е. если это фрагмент содержимого) и NULL, если его аргумент также NULL. Чем документы отличаются от фрагментов содержимого, вы можете узнать в Разделе 8.13.

9.14.2.2. `IS NOT DOCUMENT`

```
xml IS NOT DOCUMENT
```

Выражение `IS NOT DOCUMENT` возвращает false, если аргумент представляет собой правильный XML-документ, true в противном случае (т. е. если это фрагмент содержимого) и NULL, если его аргумент — NULL.

9.14.2.3. `XMLEXISTS`

```
XMLEXISTS(текст
 PASSING [BY REF] xml [BY REF])
```

Функция `xmlexists` вычисляет выражение XPath 1.0 (первый аргумент), используя в качестве элемента контекста переданное XML-значение. Эта функция возвращает false, если в результате этого вычисления выдаётся пустое множество узлов, или true, если выдаётся любое другое значение. Если один из аргументов равен NULL, результатом также будет NULL. Отличный от NULL аргумент, передающий элемент контекста, должен представлять XML-документ, а не фрагмент содержимого или какое-либо значение, недопустимое в XML.

Пример:

```
SELECT xmlexists('//town[text() = ''Toronto'']' PASSING BY REF '<towns><town>Toronto</
town><town>Ottawa</town></towns>');

 xmlexists
------------
 t
(1 row)
```

Postgres Pro принимает предложение `BY REF`, но игнорирует его, о чём рассказывается в Подразделе D.3.2. Согласно стандарту SQL, функция `xmlexists` должна вычислять выражение, используя средства языка XML Query, но Postgres Pro воспринимает только выражения XPath 1.0, что освещается в Подразделе D.3.1.

9.14.2.4. `xml_is_well_formed`

```
xml_is_well_formed(текст)
xml_is_well_formed_document(текст)
xml_is_well_formed_content(текст)
```

Эти функции проверяют, является ли `текст` правильно оформленным XML, и возвращают соответствующее логическое значение. Функция `xml_is_well_formed_document` проверяет аргумент как правильно оформленный документ, а `xml_is_well_formed_content` — правильно оформленное содержимое. Функция `xml_is_well_formed` может делать первое или второе, в зависимости от значения параметра конфигурации xmloption (`DOCUMENT` или `CONTENT`, соответственно). Это значит, что `xml_is_well_formed` помогает понять, будет ли успешным простое приведение к типу `xml`, тогда как две другие функции проверяют, будут ли успешны соответствующие варианты `XMLPARSE`.

Примеры:

```
SET xmloption TO DOCUMENT;
SELECT xml_is_well_formed('<>');
 xml_is_well_formed
--------------------
 f
(1 row)

SELECT xml_is_well_formed('<abc/>');
 xml_is_well_formed
--------------------
 t
(1 row)

SET xmloption TO CONTENT;
SELECT xml_is_well_formed('abc');
 xml_is_well_formed
--------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/
stuff">bar</pg:foo>');
 xml_is_well_formed_document
-----------------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/
stuff">bar</my:foo>');
 xml_is_well_formed_document
-----------------------------
 f
(1 row)
```

Последний пример показывает, что при проверке также учитываются сопоставления пространств имён.

9.14.3. Обработка XML

Для обработки значений типа `xml` в Postgres Pro представлены функции `xpath` и `xpath_exists`, вычисляющие выражения XPath 1.0, а также табличная функция `XMLTABLE`.

9.14.3.1. `xpath`

`xpath(`*`xpath`*`, `*`xml`*` [, `*`nsarray`*`])`

Функция `xpath` вычисляет выражение XPath 1.0 в аргументе *`xpath`* (типа `text`) для заданного *`xml`*. Она возвращает массив XML-значений с набором узлов, полученных при вычислении выражения XPath. Если выражение XPath выдаёт не набор узлов, а скалярное значение, возвращается массив из одного элемента.

Вторым аргументом должен быть правильно оформленный XML-документ. В частности, в нём должен быть единственный корневой элемент.

В необязательном третьем аргументе функции передаются сопоставления пространств имён. Эти сопоставления должны определяться в двумерном массиве типа `text`, во второй размерности которого 2 элемента (т. е. это должен быть массив массивов, состоящих из 2 элементов). В первом элементе каждого массива определяется псевдоним (префикс) пространства имён, а во

втором — его URI. Псевдонимы, определённые в этом массиве, не обязательно должны совпадать с префиксами пространств имён в самом XML-документе (другими словами, для XML-документа и функции `xpath` псевдонимы имеют *локальный* характер).

Пример:

```
SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
             ARRAY[ARRAY['my', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

Для пространства имён по умолчанию (анонимного) это выражение можно записать так:

```
SELECT xpath('//mydefns:b/text()', '<a xmlns="http://example.com"><b>test</b></a>',
             ARRAY[ARRAY['mydefns', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

9.14.3.2. xpath_exists

```
xpath_exists(xpath, xml [, nsarray])
```

Функция `xpath_exists` представляет собой специализированную форму функции `xpath`. Она возвращает не отдельные XML-значения, удовлетворяющие выражению XPath 1.0, а только один логический признак, показывающий, имеются ли такие значения (то есть выдаёт ли это выражение что-либо, отличное от пустого множества узлов). Эта функция равнозначна стандартному условию `XMLEXISTS`, за исключением того, что она также поддерживает сопоставления пространств имён.

Пример:

```
SELECT xpath_exists('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
                    ARRAY[ARRAY['my', 'http://example.com']]);

 xpath_exists
--------------
 t
(1 row)
```

9.14.3.3. xmltable

```
xmltable( [XMLNAMESPACES(URI пространства имён AS имя пространства имён[, ...]),]
          выражение_строки PASSING [BY REF] выражение_документа [BY REF]
          COLUMNS имя { тип [PATH выражение_столбца] [DEFAULT выражение_по_умолчанию]
 [NOT NULL | NULL]
                        | FOR ORDINALITY }
                  [, ...]
)
```

Функция `xmltable` строит таблицу из данного XML-значения по фильтру XPath, извлекающему строки, используя набор определений столбцов.

Необязательное предложение `XMLNAMESPACES` задаёт разделённый запятыми список пространств имён. В нём определяются пространства имён XML и их псевдонимы. Определение пространства по умолчанию в настоящее время не поддерживается.

В обязательном аргументе *выражение_строки* передаётся выражение XPath 1.0, которое будет вычисляться для передаваемого в аргументе *выражение_документа* элемента контекста и результатом которого будет набор узлов XML. Эти узлы `xmltable` преобразует в выходные строки. Если *выражение_документа* — NULL или если *выражение_строки* выдаёт пустой набор узлов или любое значение, отличное от набора узлов, выходные строки не выдаются.

Выражение_документа передаёт элемент контекста для *выражения_строки*. Таким элементом должен быть правильно оформленный XML-документ; фрагменты/наборы деревьев не допускаются. Предложение `BY REF` принимается, но игнорируется, о чём рассказывается в Подразделе D.3.2. Согласно стандарту SQL, функция `xmltable` должна вычислять выражение, используя средства языка XML Query, но Postgres Pro воспринимает только выражения XPath 1.0, что освещается в Подразделе D.3.1.

В обязательном предложении `COLUMNS` задаётся список столбцов в выходной таблице. Каждый элемент этого предложения описывает один столбец. Формат этого предложения показан выше в примере синтаксиса. Имя столбца и его тип должны задаваться обязательно: путь, значение по умолчанию и признак допустимости NULL могут опускаться.

Столбец с признаком `FOR ORDINALITY` будет заполняться номерами строк, начиная с 1, в том порядке, в котором эти строки идут в наборе узлов, представляющем результат *выражения_строки*. Признак `FOR ORDINALITY` может быть не больше чем у одного столбца.

> **Примечание**
>
> В XPath 1.0 не определён порядок узлов в наборе, поэтому код, рассчитывающий на определённый порядок результатов, будет зависеть от конкретной реализации. Подробнее об этом рассказывается в Подразделе D.3.1.2.

В *выражении_столбца* задаётся выражение XPath 1.0, которое вычисляется для каждой строки применительно к текущему узлу, полученному в результате вычисления *выражения_строки* и служащему элементом контекста, и выдаёт значение столбца. Если `выражение_столбца` отсутствует, в качестве неявного пути используется имя столбца.

Если выражение XPath возвращает не XML (это может быть строка, логическое или числовое значение в XPath 1.0) и столбец имеет тип Postgres Pro, отличный от `xml`, значение присваивается столбцу так же, как типу Postgres Pro присваивается строковое представление значения. В данной версии логический или числовой результат XPath должен явно приводиться к строковому типу (то есть исходное выражение столбца нужно обернуть в вызов функции `string` языка XPath 1.0); в этом случае Postgres Pro сможет успешно присвоить строку результирующему столбцу в SQL, имеющему логический или числовой тип. Эти правила преобразования отличаются от описанных в стандарте, о чём рассказывается в Подразделе D.3.1.3.

В текущей версии результирующие столбцы SQL, имеющие тип `xml`, или XPath-выражения столбцов, выдающие типы XML, вне зависимости от типа выходного столбца SQL, обрабатываются как описано в Подразделе D.3.2; в Postgres Pro 12 это поведение значительно изменилось.

Когда выражение пути возвращает для данной строки пустой набор узлов (обычно когда нет соответствия этому пути), столбец получает значение `NULL`, если только не задано *выражение_по_умолчанию*. Если же оно задано, столбец получает результат данного выражения.

Столбцы могут иметь признак `NOT NULL`. Если *выражение_столбца* для столбца с признаком `NOT NULL` не соответствует ничему и при этом отсутствует указание `DEFAULT` или *выражение_по_умолчанию* также выдаёт NULL, происходит ошибка.

Указанное *выражение_по_умолчанию* вычисляется не единожды для вызова функции `xmltable`, а каждый раз, когда для столбца требуется очередное значение по умолчанию. Если же это выражение оказывается стабильным или постоянным, повторное вычисление может не

выполняться. Это означает, что в *выражении_по_умолчанию* вы можете с пользой применять изменчивые функции, такие как `nextval`.

Примеры:

```
CREATE TABLE xmldata AS SELECT
xml $$
<ROWS>
  <ROW id="1">
    <COUNTRY_ID>AU</COUNTRY_ID>
    <COUNTRY_NAME>Australia</COUNTRY_NAME>
  </ROW>
  <ROW id="5">
    <COUNTRY_ID>JP</COUNTRY_ID>
    <COUNTRY_NAME>Japan</COUNTRY_NAME>
    <PREMIER_NAME>Shinzo Abe</PREMIER_NAME>
    <SIZE unit="sq_mi">145935</SIZE>
  </ROW>
  <ROW id="6">
    <COUNTRY_ID>SG</COUNTRY_ID>
    <COUNTRY_NAME>Singapore</COUNTRY_NAME>
    <SIZE unit="sq_km">697</SIZE>
  </ROW>
</ROWS>
$$ AS data;

SELECT xmltable.*
  FROM xmldata,
       XMLTABLE('//ROWS/ROW'
                PASSING data
                COLUMNS id int PATH '@id',
                        ordinality FOR ORDINALITY,
                        "COUNTRY_NAME" text,
                        country_id text PATH 'COUNTRY_ID',
                        size_sq_km float PATH 'SIZE[@unit = "sq_km"]',
                        size_other text PATH
                             'concat(SIZE[@unit!="sq_km"], " ", SIZE[@unit!="sq_km"]/
@unit)',
                        premier_name text PATH 'PREMIER_NAME' DEFAULT 'not
 specified') ;

 id | ordinality | COUNTRY_NAME | country_id | size_sq_km |  size_other  | premier_name

----+------------+--------------+------------+------------+--------------
+---------------
  1 |          1 | Australia    | AU         |            |              | not
 specified
  5 |          2 | Japan        | JP         |            | 145935 sq_mi | Shinzo Abe
  6 |          3 | Singapore    | SG         |        697 |              | not
 specified
```

Следующий пример иллюстрирует сложение нескольких узлов text(), использование имени столбца в качестве фильтра XPath и обработку пробельных символов, XML-комментариев и инструкций обработки:

```
CREATE TABLE xmlelements AS SELECT
xml $$
  <root>
   <element>  Hello<!-- xyxxz -->2a2<?aaaaa?> <!--x-->  bbb<x>xxx</x>CC  </element>
  </root>
```

```
$$ AS data;

SELECT xmltable.*
  FROM xmlelements, XMLTABLE('/root' PASSING data COLUMNS element text);
       element
----------------------
   Hello2a2   bbbCC
```

Следующий пример показывает, как с помощью предложения `XMLNAMESPACES` можно задать список пространств имён, используемых в XML-документе и в выражениях XPath:

```
WITH xmldata(data) AS (VALUES ('
<example xmlns="http://example.com/myns" xmlns:B="http://example.com/b">
 <item foo="1" B:bar="2"/>
 <item foo="3" B:bar="4"/>
 <item foo="4" B:bar="5"/>
</example>'::xml)
)
SELECT xmltable.*
  FROM XMLTABLE(XMLNAMESPACES('http://example.com/myns' AS x,
                              'http://example.com/b' AS "B"),
              '/x:example/x:item'
                 PASSING (SELECT data FROM xmldata)
                 COLUMNS foo int PATH '@foo',
                   bar int PATH '@B:bar');
 foo | bar
-----+-----
   1 |   2
   3 |   4
   4 |   5
(3 rows)
```

9.14.4. Отображение таблиц в XML

Следующие функции отображают содержимое реляционных таблиц в значения XML. Их можно рассматривать как средства экспорта в XML:

```
table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xml(query text, nulls boolean, tableforest boolean, targetns text)
cursor_to_xml(cursor refcursor, count int, nulls boolean,
              tableforest boolean, targetns text)
```

Результат всех этих функций имеет тип `xml`.

`table_to_xml` отображает в xml содержимое таблицы, имя которой задаётся в параметре *`tbl`*. Тип `regclass` принимает идентификаторы строк в обычной записи, которые могут содержать указание схемы и кавычки. Функция `query_to_xml` выполняет запрос, текст которого передаётся в параметре *`query`*, и отображает в xml результирующий набор. Последняя функция, `cursor_to_xml` выбирает указанное число строк из курсора, переданного в параметре *`cursor`*. Этот вариант рекомендуется использовать с большими таблицами, так как все эти функции создают результирующий xml в памяти.

Если параметр *`tableforest`* имеет значение false, результирующий XML-документ выглядит так:

```
<имя_таблицы>
  <row>
    <имя_столбца1> данные </имя_столбца1>
    <имя_столбца2> данные </имя_столбца2>
  </row>

  <row>
```

```
    ...
  </row>

  ...
</имя_таблицы>
```

А если *tableforest* равен true, в результате будет выведен следующий фрагмент XML:

```
<имя_таблицы>
  <имя_столбца1> данные </имя_столбца1>
  <имя_столбца2> данные </имя_столбца2>
</имя_таблицы>

<имя_таблицы>
  ...
</имя_таблицы>

...
```

Если имя таблицы неизвестно, например, при отображении результатов запроса или курсора, вместо него в первом случае вставляется `table`, а во втором — `row`.

Выбор между этими форматами остаётся за пользователем. Первый вариант позволяет создать готовый XML-документ, что может быть полезно для многих приложений, а второй удобно применять с функцией `cursor_to_xml`, если её результаты будут собираться в документ позже. Полученный результат можно изменить по вкусу с помощью рассмотренных выше функций создания XML-содержимого, в частности `xmlelement`.

Значения данных эти функции отображают так же, как и ранее описанная функция `xmlelement`.

Параметр *nulls* определяет, нужно ли включать в результат значения NULL. Если он установлен, значения NULL в столбцах представляются так:

```
<имя_столбца xsi:nil="true"/>
```

Здесь `xsi` — префикс пространства имён XML Schema Instance. При этом в результирующий XML будет добавлено соответствующее объявление пространства имён. Если же данный параметр равен false, столбцы со значениями NULL просто не будут выводиться.

Параметр *targetns* определяет целевое пространство имён для результирующего XML. Если пространство имён не нужно, значением этого параметра должна быть пустая строка.

Следующие функции выдают документы XML Schema, которые содержат схемы отображений, выполняемых соответствующими ранее рассмотренными функциями:

```
table_to_xmlschema(tbl regclass, nulls boolean, tableforest boolean,
  targetns text)
query_to_xmlschema(query text, nulls boolean, tableforest boolean,
  targetns text)
cursor_to_xmlschema(cursor refcursor, nulls boolean, tableforest boolean,
  targetns text)
```

Чтобы результаты отображения данных в XML соответствовали XML-схемам, важно, чтобы паре функций передавались одинаковые параметры.

Следующие функции выдают отображение данных в XML и соответствующую XML-схему в одном документе (или фрагменте), объединяя их вместе. Это может быть полезно там, где желательно получить самодостаточные результаты с описанием:

```
table_to_xml_and_xmlschema(tbl regclass, nulls boolean, tableforest boolean,
  targetns text)
query_to_xml_and_xmlschema(query text, nulls boolean, tableforest boolean,
  targetns text)
```

В дополнение к ним есть следующие функции, способные выдать аналогичные представления для целых схем в базе данных или даже всей текущей базы данных:

```
schema_to_xml(schema name, nulls boolean, tableforest boolean,
  targetns text)
schema_to_xmlschema(schema name, nulls boolean, tableforest boolean,
  targetns text)
schema_to_xml_and_xmlschema(schema name, nulls boolean, tableforest boolean,
  targetns text)

database_to_xml(nulls boolean, tableforest boolean, targetns text)
database_to_xmlschema(nulls boolean, tableforest boolean, targetns text)
database_to_xml_and_xmlschema(nulls boolean, tableforest boolean,
  targetns text)
```

Заметьте, что объём таких данных может быть очень большим, а XML будет создаваться в памяти. Поэтому, вместо того, чтобы пытаться отобразить в XML сразу всё содержимое больших схем или баз данных, лучше делать это по таблицам, возможно даже используя курсор.

Результат отображения содержимого схемы будет выглядеть так:

```
<имя_схемы>

отображение-таблицы1

отображение-таблицы2

...

</имя_схемы>
```

Формат отображения таблицы определяется параметром *tableforest*, описанным выше.

Результат отображения содержимого базы данных будет таким:

```
<имя_схемы>

<имя_схемы1>
  ...
</имя_схемы1>

<имя_схемы2>
  ...
</имя_схемы2>

...

</имя_схемы>
```

Здесь отображение схемы имеет вид, показанный выше.

В качестве примера, иллюстрирующего использование результата этих функций, на Примере 9.1 показано XSLT-преобразование, которое переводит результат функции `table_to_xml_and_xmlschema` в HTML-документ, содержащий таблицу с данными. Подобным образом результаты этих функций можно преобразовать и в другие форматы на базе XML.

Пример 9.1. XSLT-преобразование, переводящее результат SQL/XML в формат HTML

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
    xmlns="http://www.w3.org/1999/xhtml"
>

  <xsl:output method="xml"
      doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
      doctype-public="-//W3C/DTD XHTML 1.0 Strict//EN"
      indent="yes"/>

  <xsl:template match="/*">
    <xsl:variable name="schema" select="//xsd:schema"/>
    <xsl:variable name="tabletypename"
                  select="$schema/xsd:element[@name=name(current())]/@type"/>
    <xsl:variable name="rowtypename"
                  select="$schema/xsd:complexType[@name=$tabletypename]/xsd:sequence/
xsd:element[@name='row']/@type"/>

    <html>
      <head>
        <title><xsl:value-of select="name(current())"/></title>
      </head>
      <body>
        <table>
          <tr>
            <xsl:for-each select="$schema/xsd:complexType[@name=$rowtypename]/
xsd:sequence/xsd:element/@name">
              <th><xsl:value-of select="."/></th>
            </xsl:for-each>
          </tr>

          <xsl:for-each select="row">
            <tr>
              <xsl:for-each select="*">
                <td><xsl:value-of select="."/></td>
              </xsl:for-each>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

9.15. Функции и операторы JSON

В Таблице 9.43 показаны операторы, позволяющие работать с двумя типами данных JSON (см. Раздел 8.14).

Таблица 9.43. Операторы для типов `json` и `jsonb`

Оператор	Тип правого операнда	Описание	Пример	Результат примера
`->`	`int`	Выдаёт элемент массива JSON (по номеру от 0, отрицательные числа задают позиции с конца)	`'[{"a":"foo"}, {"b":"bar"}, {"c":"baz"}]'::json->2`	`{"c":"baz"}`

Оператор	Тип правого операнда	Описание	Пример	Результат примера
->	text	Выдаёт поле объекта JSON по ключу	'{"a": {"b":"foo"}}'::json->'a'	{"b":"foo"}
->>	int	Выдаёт элемент массива JSON в типе text	'[1,2,3]'::json->>2	3
->>	text	Выдаёт поле объекта JSON в типе text	'{"a":1,"b":2}'::json->>'b'	2
#>	text[]	Выдаёт объект JSON по заданному пути	'{"a": {"b": {"c": "foo"}}}'::json#>'{a,b}'	{"c": "foo"}
#>>	text[]	Выдаёт объект JSON по заданному пути в типе text	'{"a":[1,2,3],"b":[4,5,6]}'::json#>>'{a,2}'	3

Примечание

Эти операторы существуют в двух вариациях для типов json и jsonb. Операторы извлечения поля/элемента/пути возвращают тот же тип, что у операнда слева (json или jsonb), за исключением тех, что возвращают тип text (они возвращают значение как текстовое). Если входные данные JSON не содержат структуры, удовлетворяющей запросу, например в них нет искомого элемента, то операторы извлечения поля/элемента/пути не выдают ошибку, а возвращают NULL. Все операторы извлечения поля/элемента/пути, принимающие целочисленные позиции в массивах JSON, поддерживают и отсчёт от конца массива по отрицательной позиции.

Стандартные операторы сравнения, приведённые в Таблице 9.1, есть для типа jsonb, но не для json. Они следуют правилам сортировки для операций B-дерева, описанным в Подразделе 8.14.4. В Разделе 9.20 вы также можете узнать об агрегатной функции json_agg, которая агрегирует значения записи в виде JSON, и агрегатной функции json_object_agg, агрегирующей пары значений в объект JSON, а также их аналогах для jsonb, функциях jsonb_agg и jsonb_object_agg.

Некоторые из следующих операторов существуют только для jsonb, как показано в Таблице 9.44. Многие из этих операторов могут быть проиндексированы с помощью классов операторов jsonb. Полное описание проверок на вхождение и существование для jsonb приведено в Подразделе 8.14.3. Как эти операторы могут использоваться для эффективного индексирования данных jsonb, описано в Подразделе 8.14.4.

Таблица 9.44. Дополнительные операторы jsonb

Оператор	Тип правого операнда	Описание	Пример
@>	jsonb	Левое значение JSON содержит на верхнем уровне путь/значение JSON справа?	'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb
<@	jsonb	Путь/значение JSON слева содержится на	'{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb

Оператор	Тип правого операнда	Описание	Пример
		верхнем уровне в правом значении JSON?	
?	text	Присутствует ли *строка* в качестве ключа верхнего уровня в значении JSON?	'{"a":1, "b":2}'::jsonb ? 'b'
?|	text[]	Какие-либо *строки* массива присутствуют в качестве ключей верхнего уровня?	'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'c']
?&	text[]	Все *строки* массива присутствуют в качестве ключей верхнего уровня?	'["a", "b"]'::jsonb ?& array['a', 'b']
||	jsonb	Соединяет два значения jsonb в новое значение jsonb	'["a", "b"]'::jsonb || '["c", "d"]'::jsonb
-	text	Удаляет пару ключ/значение или элемент-*строку* из левого операнда. Пары ключ/значение выбираются по значению ключа.	'{"a": "b"}'::jsonb - 'a'
-	text[]	Удаляет множество пар ключ/значение или элементы-*строки* из левого операнда. Пары ключ/значение выбираются по значению ключа.	'{"a": "b", "c": "d"}'::jsonb - '{a, c}'::text[]
-	integer	Удаляет из массива элемент в заданной позиции (отрицательные номера позиций отсчитываются от конца). Выдаёт ошибку, если контейнер верхнего уровня — не массив.	'["a", "b"]'::jsonb - 1
#-	text[]	Удаляет поле или элемент с заданным путём (для массивов JSON отрицательные номера позиций отсчитываются от конца)	'["a", {"b":1}]'::jsonb #- '{1,b}'

Примечание

Оператор || соединяет два объекта JSON вместе, формируя объект с объединённым набором ключей и значений, при этом в случае совпадения ключей выбирается значение

из второго объекта. С любыми другими аргументами формируется массив JSON: сначала любой аргумент, отличный от массива, преобразуется в одноэлементный массив, а затем два полученных массива соединяются. Эта операция не рекурсивна — объединение производится только на верхнем уровне структуры объекта или массива.

В Таблице 9.45 показаны функции, позволяющие создавать значения типов `json` и `jsonb`. (Для типа `jsonb` нет аналогов функций `row_to_json` и `array_to_json`, но практически тот же результат можно получить с помощью `to_jsonb`.)

Таблица 9.45. Функции для создания JSON

Функция	Описание	Пример	Результат примера
`to_json(anyelement)` `to_jsonb( anyelement)`	Возвращает значение в виде `json` или `jsonb`. Массивы и составные структуры преобразуются (рекурсивно) в массивы и объекты; для других типов, для которых определено приведение к `json`, применяется эта функция приведения, а для всех остальных выдаётся скалярное значение. Значения всех скалярных типов, кроме числового, логического и NULL, представляются в текстовом виде, в стиле, допустимом для значений `json` или `jsonb`.	`to_json('Fred said "Hi."'::text)`	`"Fred said \"Hi.\""`
`array_to_json( anyarray [, pretty_ bool])`	Возвращает массив в виде массива JSON. Многомерный массив Postgres Pro становится массивом массивов JSON. Если параметр *pretty_ bool* равен true, между элементами 1-ой размерности вставляются разрывы строк.	`array_to_json( '{{1,5},{99, 100}}'::int[])`	`[[1,5],[99,100]]`
`row_to_json(record [, pretty_bool])`	Возвращает кортеж в виде объекта JSON. Если параметр *pretty_ bool* равен true, между элементами 1-ой размерности вставляются разрывы строк.	`row_to_json(row( 1,'foo'))`	`{"f1":1,"f2":"foo"}`

Функция	Описание	Пример	Результат примера
`json_build_array( VARIADIC "any")` `jsonb_build_array( VARIADIC "any")`	Формирует массив JSON (возможно, разнородный) из переменного списка аргументов.	`json_build_array( 1,2,'3',4,5)`	`[1, 2, "3", 4, 5]`
`json_build_object( VARIADIC "any")` `jsonb_build_object( VARIADIC "any")`	Формирует объект JSON из переменного списка аргументов. По соглашению в этом списке перечисляются по очереди ключи и значения.	`json_build_object( 'foo',1,'bar',2)`	`{"foo": 1, "bar": 2}`
`json_object(text[])` `jsonb_object( text[])`	Формирует объект JSON из текстового массива. Этот массив должен иметь либо одну размерность с чётным числом элементов (в этом случае они воспринимаются как чередующиеся ключи/значения), либо две размерности и при этом каждый внутренний массив содержит ровно два элемента, которые воспринимаются как пара ключ/значение.	`json_object('{a, 1, b, "def", c, 3.5}')` `json_object('{{a, 1},{b, "def"}, {c, 3.5}}')`	`{"a": "1", "b": "def", "c": "3.5"}`
`json_object(keys text[], values text[])` `jsonb_object(keys text[], values text[])`	Эта форма `json_object` принимает ключи и значения по парам из двух отдельных массивов. Во всех остальных отношениях она не отличается от формы с одним аргументом.	`json_object('{a, b}', '{1,2}')`	`{"a": "1", "b": "2"}`

Примечание

Функции `array_to_json` и `row_to_json` подобны `to_json`, но предлагают возможность улучшенного вывода. Действие `to_json`, описанное выше, распространяется на каждое отдельное значение, преобразуемое этими функциями.

Примечание

В расширении hstore определено преобразование из `hstore` в `json`, так что значения `hstore`, преобразуемые функциями создания JSON, будут представлены в виде объектов JSON, а не как примитивные строковые значения.

В Таблице 9.46 показаны функции, предназначенные для работы со значениями `json` и `jsonb`.

Таблица 9.46. Функции для обработки JSON

Функция	Тип результата	Описание	Пример	Результат примера
`json_array_length(json)` `jsonb_array_length(jsonb)`	`int`	Возвращает число элементов во внешнем массиве JSON.	`json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]')`	`5`
`json_each(json)` `jsonb_each(jsonb)`	`setof key text, value json` `setof key text, value jsonb`	Разворачивает внешний объект JSON в набор пар ключ/значение (key/value).	`select * from json_each('{"a":"foo", "b":"bar"}')`	`key | value` `-----+-------` `a | "foo"` `b | "bar"`
`json_each_text(json)` `jsonb_each_text(jsonb)`	`setof key text, value text`	Разворачивает внешний объект JSON в набор пар ключ/значение (key/value). Возвращаемые значения будут иметь тип `text`.	`select * from json_each_text('{"a":"foo", "b":"bar"}')`	`key | value` `-----+-------` `a | foo` `b | bar`
`json_extract_path(from_json json, VARIADIC path_elems text[])` `jsonb_extract_path(from_json jsonb, VARIADIC path_elems text[])`	`json` `jsonb`	Возвращает значение JSON по пути, заданному элементами пути (*path_elems*) (равнозначно оператору `#>` operator).	`json_extract_path('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}', 'f4')`	`{"f5":99,"f6":"foo"}`
`json_extract_path_text(from_json json, VARIADIC path_elems text[])` `jsonb_extract_path_text(from_json jsonb, VARIADIC path_elems text[])`	`text`	Возвращает значение JSON по пути, заданному элементами пути *path_elems*, как `text` (равнозначно оператору `#>>`).	`json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}', 'f4', 'f6')`	`foo`
`json_object_keys(json)` `jsonb_object_keys(jsonb)`	`setof text`	Возвращает набор ключей во внешнем объекте JSON.	`json_object_keys('{"f1":"abc","f2":{"f3":"a","f4":"b"}}')`	`json_object_keys` `------------------` `f1` `f2`
`json_populate_record(base anyelement, from_json json)`	`anyelement`	Разворачивает объект из *from_json* в табличную строку, в которой столбцы	`create type myrowtype as (a int, b json, c json);`	`a | b | c` `---+---+---`

Функция	Тип результата	Описание	Пример	Результат примера
jsonb_populate_record(base anyelement, from_json jsonb)		соответствуют типу строки, заданному параметром *base* (см. примечания ниже).	select * from json_populate_record(null::myrowtype, '{"a": 1, "b": ["2", "a b"], "c": {"d": 4, "e": "a b c"}}')	1 | {2,"a b"} | (4,"a b c")
json_populate_recordset(base anyelement, from_json json) jsonb_populate_recordset(base anyelement, from_json jsonb)	setof anyelement	Разворачивает внешний массив объектов из *from_json* в набор табличных строк, в котором столбцы соответствуют типу строки, заданному параметром *base* (см. примечания ниже).	create type myrowtype as (a int, b int); select * from json_populate_recordset(null::myrowtype, '[{"a":1, "b":2},{"a":3, "b":4}]') as (a int, b int)	a | b ---+--- 1 | 2 3 | 4
json_array_elements(json) jsonb_array_elements(jsonb)	setof json setof jsonb	Разворачивает массив JSON в набор значений JSON.	select * from json_array_elements('[1, true, [2, false]]')	value ----------- 1 true [2,false]
json_array_elements_text(json) jsonb_array_elements_text(jsonb)	setof text	Разворачивает массив JSON в набор значений text.	select * from json_array_elements_text('["foo", "bar"]')	value ----------- foo bar
json_typeof(json) jsonb_typeof(jsonb)	text	Возвращает тип внешнего значения JSON в виде текстовой строки. Возможные типы: object, array, string, number, boolean и null.	json_typeof('-123.4')	number
json_to_record(json) jsonb_to_record(jsonb)	record	Формирует обычную запись из объекта JSON (см. примечания ниже). Как и со всеми функциями, возвращающими record, при вызове необходимо явно	create type myrowtype as (a int, b text); select * from json_to_record('{"a":1, "b":[1,2,3],	a | b | c | d | r ---+---+---+---+--- 1 | [1,2,3] | {1,2,3} | | (123,"a b c")

Функция	**Тип результата**	**Описание**	**Пример**	**Результат примера**
		определить структуру записи с помощью предложения AS.	"c":[1,2,3], "e":"bar","r": {"a": 123, "b": "a b c"}}') as x(a int, b text, c int[], d text, r myrowtype)	
json_to_ recordset(json) jsonb_to_ recordset(jsonb)	setof record	Формирует обычный набор записей из массива объекта JSON (см. примечания ниже). Как и со всеми функциями, возвращающими record, при вызове необходимо явно определить структуру записи с помощью предложения AS.	select * from json_to_ recordset('[{"a":1, "b":"foo"}, {"a":"2", "c":"bar"}]') as x(a int, b text);	a | b ---+----- 1 | foo 2 |
json_strip_ nulls(from_ json json) jsonb_strip_ nulls(from_ json jsonb)	json jsonb	Возвращает значение *from_ json* , из которого исключаются все поля объекта, содержащие значения NULL. Другие значения NULL остаются нетронутыми.	json_strip_ nulls('[{"f1":1, "f2":null},2, null,3]')	[{"f1":1},2, null,3]
jsonb_set(target jsonb, path text[], new_value jsonb [, create_missing boolean])	jsonb	Возвращает значение *target*, в котором раздел с заданным путём (*path*) заменяется новым значением (*new_value*), либо в него добавляется значение *new_ value* , если аргумент *create_ missing* равен true (это значение по умолчанию) и элемент, на который ссылается *path*, не существует. Как и с операторами, рассчитанными	jsonb_set('[{"f1":1, "f2":null},2, null,3]', '{0, f1}','[2,3, 4]', false) jsonb_set('[{"f1":1, "f2":null}, 2]', '{0,f3}', '[2,3,4]')	[{"f1":[2,3, 4],"f2":null}, 2,null,3] [{"f1": 1, "f2": null, "f3": [2, 3, 4]}, 2]

Функция	**Тип результата**	**Описание**	**Пример**	**Результат примера**
		на пути, отрицательные числа в пути (*path*) обозначают отсчёт от конца массивов JSON.		
`jsonb_insert( target jsonb, path text[], new_value jsonb [, insert_after boolean])`	`jsonb`	Возвращает значение *target* с вставленным в него новым значением *new_value* . Если место в *target*, выбранное путём *path*, оказывается в массиве JSONB, *new_value* будет вставлен до (по умолчанию) или после (если параметр *insert_after* равен true) выбранной позиции. Если место в *target*, выбранное путём *path*, оказывается в объекте JSONB, значение *new_value* будет вставлено в него, только если заданный путь *path* не существует. Как и с операторами, рассчитанными на пути, отрицательные числа в пути (*path*) обозначают отсчёт от конца массивов JSON.	`jsonb_insert( '{"a": [0,1, 2]}', '{a, 1}', '"new_value"')` `jsonb_insert( '{"a": [0,1, 2]}', '{a, 1}', '"new_value"', true)`	`{"a": [0, "new_value", 1, 2]}` `{"a": [0, 1, "new_value", 2]}`
`jsonb_pretty( from_json jsonb)`	`text`	Возвращает значение *from_json* в виде текста JSON с отступами.	`jsonb_pretty( '[{"f1":1, "f2":null},2, null,3]')`	`[ { "f1": 1, "f2": null }, 2, null, 3 ]`

Примечание

Многие из этих функций и операторов преобразуют спецпоследовательности Unicode в JSON-строках в соответствующие одиночные символы. Для входных данных типа `jsonb` это ничем не грозит, так как преобразование уже выполнено; однако для типа `json` в результате может произойти ошибка, как отмечено в Разделе 8.14.

Примечание

Функции `json[b]_populate_record`, `json[b]_populate_recordset`, `json[b]_to_record` и `json[b]_to_recordset` принимают объект JSON или массив объектов и извлекают значения, связанные с ключами, по именам, соответствующим именам столбцов выходного типа. Поля объектов, не соответствующие никаким именам выходных столбцов, игнорируются, а выходные столбцы, для которых не находятся поля в объекте, заполняются значениями NULL. Для преобразования значения JSON в тип SQL выходного столбца последовательно применяются следующие правила:

- Значение NULL в JSON всегда преобразуется в SQL NULL.
- Если выходной столбец имеет тип `json` или `jsonb`, значение JSON воспроизводится без изменений.
- Если выходной столбец имеет составной тип (тип кортежа) и значение JSON является объектом JSON, поля этого объекта преобразуются в столбцы типа выходного кортежа в результате рекурсивного применения этих правил.
- Подобным образом, если выходной столбец имеет тип-массив и значение JSON представляет массив JSON, элементы данного массива преобразуются в элементы выходного массива в результате рекурсивного применения этих правил.
- Если же значение JSON является строковой константой, содержимое этой строки передаётся входной функции преобразования для типа данных целевого столбца.
- В противном случае функции преобразования для типа данных целевого столбца передаётся обычное текстовое представление значения JSON.

Хотя в примерах использования перечисленных функций применяются константы, обычно эти функции обращаются к таблицам в предложении `FROM`, а в качестве аргумента указывается один из столбцов типа `json` или `jsonb`. Извлечённые значения затем могут использоваться в других частях запроса, например, в предложениях `WHERE` и результирующих списках. Извлекая множество значений подобным образом, можно значительно увеличить производительность по сравнению с использованием операторов, работающих с отдельными ключами.

Примечание

В `target` должны присутствовать все элементы пути, заданного параметром `path` функций `jsonb_set` и `jsonb_insert`, за исключением последнего. Если `create_missing` равен false, должны присутствовать абсолютно все элементы пути `path`, переданного функции `jsonb_set`. Если это условие не выполняется, значение `target` возвращается неизменённым.

Если последним элементом пути оказывается ключ объекта, он будет создан в случае отсутствия и получит новое значение. Если последний элемент пути — позиция в массиве, то когда она положительна, целевой элемент отсчитывается слева, а когда отрицательна — справа, то есть `-1` указывает на самый правый элемент и т. д. Если позиция лежит вне диапазона -длина_массива .. длина_массива -1, и параметр create_missing равен true, новое значение добавляется в начало массива, если позиция отрицательна, и в конец, если положительна.

> **Примечание**
>
> Значение `null`, возвращаемое функцией `json_typeof`, не следует путать с SQL NULL. Тогда как при вызове `json_typeof('null'::json)` возвращается `null`, при вызове `json_typeof(NULL::json)` будет возвращено значение SQL NULL.

> **Примечание**
>
> Если аргумент функции `json_strip_nulls` содержит повторяющиеся имена полей в любом объекте, в результате могут проявиться семантические различия, в зависимости от порядка этих полей. Это не проблема для функции `jsonb_strip_nulls`, так как в значениях `jsonb` имена полей не могут дублироваться.

9.16. Функции для работы с последовательностями

В этом разделе описаны функции для работы с объектами, представляющими *последовательности*. Такие объекты (также называемыми генераторами последовательностей или просто последовательностями) являются специальными таблицами из одной строки и создаются командой CREATE SEQUENCE. Используются они обычно для получения уникальных идентификаторов строк таблицы. Функции, перечисленные в Таблице 9.47, предоставляют простые и безопасные для параллельного использования методы получения очередных значений таких последовательностей.

Таблица 9.47. Функции для работы с последовательностями

Функция	Тип результата	Описание
`currval( regclass)`	`bigint`	Выдаёт значение заданной последовательности, которое было возвращено при последнем вызове функции `nextval`
`lastval()`	`bigint`	Выдаёт значение любой последовательности, которое было возвращено при последнем вызове функции `nextval`
`nextval( regclass)`	`bigint`	Продвигает последовательность к следующему значению и возвращает его
`setval( regclass, bigint)`	`bigint`	Устанавливает текущее значение последовательности
`setval( regclass, bigint, boolean)`	`bigint`	Устанавливает текущее значение последовательности и флаг `is_called` , указывающий на то, что это значение использовалось

Последовательность, к которой будет обращаться одна из этих функций, определяется аргументом `regclass`, задающим просто OID последовательности в системном каталоге `pg_class`. Вычислять этот OID вручную не нужно, так как процедура ввода данных `regclass` автоматически выполнит эту работу за вас. Просто запишите имя последовательности в апострофах, чтобы оно выглядело как строковая константа. Для совместимости с обычными именами SQL эта строка будет переведена в нижний регистр, если только она не заключена в кавычки. Например:

```
nextval('foo')      обращается к последовательности foo
nextval('FOO')      обращается к последовательности foo
nextval('"Foo"')    обращается к последовательности Foo
```

При необходимости имя последовательности можно дополнить именем схемы:

```
nextval('myschema.foo')     обращается к myschema.foo
nextval('"myschema".foo')   то же самое
nextval('foo')              ищет foo в пути поиска
```

Подробнее тип `regclass` описан в Разделе 8.18.

> **Примечание**
>
> В PostgreSQL до версии 8.1 аргументы этих функций имели тип `text`, а не `regclass`, и поэтому описанное выше преобразование текстовой строки в OID имело место при каждом вызове функции. Это поведение сохраняется и сейчас для обратной совместимости, но сейчас оно реализовано как неявное приведение типа `text` к типу `regclass` перед вызовом функции.
>
> Когда вы записываете аргумент функции, работающей с последовательностью, как текстовую строку в чистом виде, она становится константой типа `regclass`. Так как фактически это будет просто значение OID, оно будет привязано к изначально идентифицированной последовательности, несмотря на то, что она может быть переименована, перенесена в другую схему и т. д. Такое «раннее связывание» обычно желательно для ссылок на последовательности в значениях столбцов по умолчанию и представлениях. Но иногда возникает необходимость в «позднем связывании», когда ссылки на последовательности распознаются в процессе выполнения. Чтобы получить такое поведение, нужно принудительно изменить тип константы с `regclass` на `text`:
>
> ```
> nextval('foo'::text)      foo распознаётся во время выполнения
> ```
>
> Заметьте, что версии PostgreSQL до 8.1 поддерживали только позднее связывание, так что это может быть полезно и для совместимости со старыми приложениями.
>
> Конечно же, аргументом таких функций может быть не только константа, но и выражение. Если это выражение текстового типа, неявное приведение типов повлечёт разрешение имени во время выполнения.

Ниже описаны все функции, предназначенные для работы с последовательностями:

`nextval`

Продвигает последовательность к следующему значению и возвращает его. Это атомарная операция: если `nextval` вызывается одновременно в нескольких сеансах, в результате каждого вызова будут гарантированно получены разные значения.

Если последовательность создаётся с параметрами по умолчанию, успешные вызовы `nextval` получают очередные значения по возрастанию, начиная с 1. Другое поведение можно получить с помощью специальных параметров в команде CREATE SEQUENCE; подробнее это описано на странице описания команды.

Этой функции требуется право `USAGE` или `UPDATE` для последовательности.

`currval`

Возвращает значение, выданное при последнем вызове `nextval` для этой последовательности в текущем сеансе. (Если в данном сеансе `nextval` ни разу не вызывалась для данной последовательности, возвращается ошибка.) Так как это значение ограничено рамками сеанса, эта функция выдаёт предсказуемый результат вне зависимости от того, вызвалась ли впоследствии `nextval` в других сеансах или нет.

Этой функции требуется право `USAGE` или `SELECT` для последовательности.

`lastval`

Возвращает значение, выданное при последнем вызове `nextval` в текущем сеансе. Эта функция подобна `currval`, но она не принимает в параметрах имя последовательности, а обращается к той последовательности, для которой вызывалась `nextval` в последний раз в текущем сеансе. Если в текущем сеансе функция `nextval` ещё не вызывалась, при вызове `lastval` произойдёт ошибка.

Этой функции требуется право `USAGE` или `SELECT` для последней использованной последовательности.

`setval`

Сбрасывает счётчик последовательности. В форме с двумя параметрами устанавливает для последовательности заданное значение поля `last_value` и значение `true` для флага `is_called`, показывающего, что при следующем вызове `nextval` последовательность должна сначала продвинуться к очередному значению, которое будет возвращено. При этом `currval` также возвратит заданное значение. В форме с тремя параметрами флагу `is_called` можно присвоить `true` или `false`. Со значением `true` она действует так же, как и форма с двумя параметрами. Если же присвоить этому флагу значение `false`, первый вызов `nextval` после этого вернёт именно заданное значение, а продвижение последовательности произойдёт при последующем вызове `nextval`. Кроме того, значение, возвращаемое `currval` в этом случае, не меняется. Например,

```
SELECT setval('foo', 42);           Следующий вызов nextval вернёт 43
SELECT setval('foo', 42, true);     То же самое
SELECT setval('foo', 42, false);    Следующий вызов nextval вернёт 42
```

Результатом самой функции `setval` будет просто значение её второго аргумента.

Этой функции требуется право `UPDATE` для последовательности.

Внимание

Во избежание блокирования параллельных транзакций, пытающихся получить значения одной последовательности, значение, выданное функцией `nextval`, не высвобождается для повторного использования, если вызывающая транзакция впоследствии прерывается. Это означает, что в случае сбоев базы данных или прерывания транзакций в последовательности задействованных значений могут появляться пропуски. Пропуски могут появиться и без прерывания транзакции. Например, команда `INSERT` с предложением `ON CONFLICT` вычислит кортеж, претендующий на добавление, произведя все требуемые вызовы `nextval`, прежде чем выявит конфликты, которые могут привести к отработке правил `ON CONFLICT` вместо добавления. Таким образом, объекты последовательностей PostgreSQL *не годятся для получения непрерывных последовательностей*.

В том же ключе изменения состояния последовательности, произведённые функцией `setval`, сразу же становятся видимыми для других транзакций и не отменяются при откате транзакции, вызвавшей эту функцию.

Если кластер базы прерывает работу до фиксации транзакции, содержащей вызов функции `nextval` или `setval`, изменение состояния последовательности может не попасть в постоянное хранилище, поэтому неизвестно, какое состояние будет иметь последовательность после перезапуска кластера: исходное или изменённое. Это не создаёт проблем при использовании последовательности внутри базы данных, так как другие изменения незафиксированных транзакций также не будут видны. Однако если значение последовательности будет сохраняться и использоваться вне базы данных, следует предварительно убедиться в том, что вызов функции `nextval` успешно зафиксирован.

9.17. Условные выражения

В этом разделе описаны SQL-совместимые условные выражения, которые поддерживаются в Postgres Pro.

> **Подсказка**
>
> Если возможностей этих условных выражений оказывается недостаточно, вероятно, имеет смысл перейти к написанию хранимых процедур на более мощном языке программирования.

9.17.1. CASE

Выражение CASE в SQL представляет собой общее условное выражение, напоминающее операторы if/else в других языках программирования:

```
CASE WHEN условие THEN результат
     [WHEN ...]
     [ELSE результат]
END
```

Предложения CASE можно использовать везде, где допускаются выражения. Каждое *условие* в нём представляет собой выражение, возвращающее результат типа boolean. Если результатом выражения оказывается true, значением выражения CASE становится *результат*, следующий за условием, а остальная часть выражения CASE не вычисляется. Если же условие не выполняется, за ним таким же образом проверяются все последующие предложения WHEN. Если не выполняется ни одно из *условий* WHEN, значением CASE становится *результат*, записанный в предложении ELSE. Если при этом предложение ELSE отсутствует, результатом выражения будет NULL.

Пример:

```
SELECT * FROM test;

 a
---
 1
 2
 3


SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

Типы данных всех выражений *результатов* должны приводиться к одному выходному типу. Подробнее это описано в Разделе 10.5.

Существует также «простая» форма выражения CASE, разновидность вышеприведённой общей формы:

```
CASE выражение
    WHEN значение THEN результат
    [WHEN ...]
    [ELSE результат]
```

```
END
```

В такой форме сначала вычисляется первое *выражение*, а затем его результат сравнивается с выражениями *значений* в предложениях WHEN, пока не будет найдено равное ему. Если такого не значения не находится, возвращается *результат* предложения ELSE (или NULL). Эта форма больше похожа на оператор switch, существующий в языке C.

Показанный ранее пример можно записать по-другому, используя простую форму CASE:

```
SELECT a,
       CASE a WHEN 1 THEN 'one'
              WHEN 2 THEN 'two'
              ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

В выражении CASE вычисляются только те подвыражения, которые необходимы для получения результата. Например, так можно избежать ошибки деления на ноль:

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

> **Примечание**
>
> Как было описано в Подразделе 4.2.14, всё же возможны ситуации, когда подвыражения вычисляются на разных этапах, так что железной гарантии, что в «CASE вычисляются только необходимые подвыражения», в принципе нет. Например, константное подвыражение 1/0 обычно вызывает ошибку деления на ноль на этапе планирования, хотя эта ветвь CASE может вовсе не вычисляться во время выполнения.

9.17.2. COALESCE

```
COALESCE(значение [, ...])
```

Функция COALESCE возвращает первый попавшийся аргумент, отличный от NULL. Если же все аргументы равны NULL, результатом тоже будет NULL. Это часто используется при отображении данных для подстановки некоторого значения по умолчанию вместо значений NULL:

```
SELECT COALESCE(description, short_description, '(none)') ...
```

Этот запрос вернёт значение description, если оно не равно NULL, либо short_description, если оно не NULL, и строку (none), если оба эти значения равны NULL.

Аргументы должны быть приводимыми к одному общему типу, который и будет типом результата (подробнее об этом говорится в Разделе 10.5).

Как и выражение CASE, COALESCE вычисляет только те аргументы, которые необходимы для получения результата; то есть, аргументы правее первого отличного от NULL аргумента не вычисляются. Эта функция соответствует стандарту SQL, а в некоторых других СУБД её аналоги называются NVL и IFNULL.

9.17.3. NULLIF

```
NULLIF(значение1, значение2)
```

Функция NULLIF выдаёт значение NULL, если *значение1* равно *значение2*; в противном случае она возвращает *значение1*. Это может быть полезно для реализации обратной операции к COALESCE. В частности, для примера, показанного выше:

```
SELECT NULLIF(value, '(none)') ...
```

В данном примере если `value` равно `(none)`, выдаётся null, а иначе возвращается значение `value`.

Два её аргумента должны быть сравнимых типов. Если говорить точнее, они сравниваются точно так же, как сравнивались бы в записи *значение1 = значение2*, так что для этих типов должен существовать подходящий оператор =.

Результат будет иметь тот же тип, что и первый аргумент, но есть одна тонкость. Эта функция фактически возвращает первый аргумент подразумеваемого оператора =, который в некоторых случаях преобразуется к типу второго аргумента. Например, `NULLIF(1, 2.2)` возвращает `numeric`, так как оператор `integer = numeric` не существует, существует только оператор `numeric = numeric`.

9.17.4. GREATEST и LEAST

`GREATEST(`*значение* `[, ...])`

`LEAST(`*значение* `[, ...])`

Функции `GREATEST` и `LEAST` выбирают наибольшее или наименьшее значение из списка выражений. Все эти выражения должны приводиться к общему типу данных, который станет типом результата (подробнее об этом в Разделе 10.5). Значения NULL в этом списке игнорируются, так что результат выражения будет равен NULL, только если все его аргументы равны NULL.

Заметьте, что функции `GREATEST` и `LEAST` не описаны в стандарте SQL, но часто реализуются в СУБД как расширения. В некоторых других СУБД они могут возвращать NULL, когда не все, а любой из аргументов равен NULL.

9.18. Функции и операторы для работы с массивами

В Таблице 9.48 показаны операторы, предназначенные для работы с массивами.

Таблица 9.48. Операторы для работы с массивами

Оператор	Описание	Пример	Результат
=	равно	`ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3]`	t
<>	не равно	`ARRAY[1,2,3] <> ARRAY[1,2,4]`	t
<	меньше	`ARRAY[1,2,3] < ARRAY[1,2,4]`	t
>	больше	`ARRAY[1,4,3] > ARRAY[1,2,4]`	t
<=	меньше или равно	`ARRAY[1,2,3] <= ARRAY[1,2,3]`	t
>=	больше или равно	`ARRAY[1,4,3] >= ARRAY[1,4,3]`	t
@>	содержит	`ARRAY[1,4,3] @> ARRAY[3,1,3]`	t
<@	содержится в	`ARRAY[2,2,7] <@ ARRAY[1,7,4,2,6]`	t
&&	пересечение (есть общие элементы)	`ARRAY[1,4,3] && ARRAY[2,1]`	t
||	соединение массива с массивом	`ARRAY[1,2,3] || ARRAY[4,5,6]`	{1,2,3,4,5,6}

Оператор	Описание	Пример	Результат
`||`	соединение массива с массивом	`ARRAY[1,2,3] || ARRAY[[4,5,6], [7,8,9]]`	`{{1,2,3},{4,5,6},{7,8,9}}`
`||`	соединение элемента с массивом	`3 || ARRAY[4,5,6]`	`{3,4,5,6}`
`||`	соединение массива с элементом	`ARRAY[4,5,6] || 7`	`{4,5,6,7}`

Операторы упорядочивания массивов (`<`, `>=` и т. д.) сравнивают содержимое массивов по элементам, используя при этом функцию сравнения для B-дерева, определённую для типа данного элемента по умолчанию, и сортируют их по первому различию. В многомерных массивах элементы просматриваются по строкам (индекс последней размерности меняется в первую очередь). Если содержимое двух массивов совпадает, а размерности различаются, результат их сравнения будет определяться первым отличием в размерностях. (В PostgreSQL до версии 8.2 поведение было другим: два массива с одинаковым содержимом считались одинаковыми, даже если число их размерностей и границы индексов различались.)

Операторы вложенности массивов (`<@` и `@>`) считают один массив вложенным в другой, если каждый элемент первого встречается во втором. Повторяющиеся элементы рассматриваются на общих основаниях, поэтому массивы `ARRAY[1]` и `ARRAY[1,1]` считаются вложенным друг в друга.

Подробнее поведение операторов с массивами описано в Разделе 8.15. За дополнительными сведениями об операторах, поддерживающих индексы, обратитесь к Разделу 11.2.

В Таблице 9.49 перечислены функции, предназначенные для работы с массивами. Дополнительная информация о них и примеры использования приведены в Разделе 8.15.

Таблица 9.49. Функции для работы с массивами

Функция	Тип результата	Описание	Пример	Результат
`array_append ( anyarray, anyelement)`	`anyarray`	добавляет элемент в конец массива	`array_append( ARRAY[1,2], 3)`	`{1,2,3}`
`array_cat ( anyarray, anyarray)`	`anyarray`	соединяет два массива	`array_cat( ARRAY[1,2,3], ARRAY[4,5])`	`{1,2,3,4,5}`
`array_ndims ( anyarray)`	`int`	возвращает число размерностей массива	`array_ndims( ARRAY[[1,2, 3], [4,5,6]])`	`2`
`array_dims ( anyarray)`	`text`	возвращает текстовое представление размерностей массива	`array_dims( ARRAY[[1,2, 3], [4,5,6]])`	`[1:2][1:3]`
`array_fill ( anyelement, int[] [, int[]])`	`anyarray`	возвращает массив, заполненный заданным значением и имеющий указанные размерности, в которых нижняя граница может быть отлична от 1	`array_fill(7, ARRAY[3], ARRAY[2])`	`[2:4]={7,7,7}`

Функция	Тип результата	Описание	Пример	Результат
array_length (anyarray, int)	int	возвращает длину указанной размерности массива	array_length(array[1,2,3], 1)	3
array_lower (anyarray, int)	int	возвращает нижнюю границу указанной размерности массива	array_lower('[0:2]={1,2, 3}'::int[], 1)	0
array_position (anyarray, anyelement [, int])	int	возвращает позицию первого вхождения второго аргумента в массиве, начиная с элемента, выбираемого третьим аргументом, либо с первого элемента (массив должен быть одномерным)	array_position(ARRAY['sun', 'mon','tue', 'wed','thu', 'fri','sat'], 'mon')	2
array_ positions (anyarray, anyelement)	int[]	возвращает массив с позициями всех вхождений второго аргумента в массиве, задаваемым первым аргументом (массив должен быть одномерным)	array_ positions(ARRAY['A','A', 'B','A'], 'A')	{1,2,4}
array_prepend (anyelement, anyarray)	anyarray	вставляет элемент в начало массива	array_prepend(1, ARRAY[2,3])	{1,2,3}
array_remove (anyarray, anyelement)	anyarray	удаляет из массива все элементы, равные заданному значению (массив должен быть одномерным)	array_remove(ARRAY[1,2,3, 2], 2)	{1,3}
array_replace (anyarray, anyelement, anyelement)	anyarray	заменяет в массиве все элементы, равные заданному значению, другим значением	array_replace(ARRAY[1,2,5, 4], 5, 3)	{1,2,3,4}
array_to_ string (	text	выводит элементы массива через заданный	array_to_ string(ARRAY[1, 2, 3,	1,2,3,*,5

Функция	Тип результата	Описание	Пример	Результат
anyarray, text [, text])		разделитель и позволяет определить замену для значения NULL	NULL, 5], ', ', '*')	
array_upper (anyarray, int)	int	возвращает верхнюю границу указанной размерности массива	array_upper(ARRAY[1,8,3, 7], 1)	4
cardinality(anyarray)	int	возвращает общее число элементов в массиве, либо 0, если массив пуст	cardinality(ARRAY[[1,2], [3,4]])	4
string_to_ array (text, text [, text])	text[]	разбивает строку на элементы массива, используя заданный разделитель и, возможно, замену для значений NULL	string_to_ array('xx~^~yy~ ^~zz', '~^~', 'yy')	{xx,NULL,zz}
unnest(anyarray)	setof anyelement	разворачивает массив в набор строк	unnest(ARRAY[1, 2])	1 2 (2 строки)
unnest(anyarray, anyarray [, ...])	setof anyelement, anyelement [, ...]	разворачивает массивы (возможно разных типов) в набор строк. Это допускается только в предложении FROM; см. Подраздел 7.2.1.4	unnest(ARRAY[1, 2], ARRAY['foo', 'bar','baz'])	1 foo 2 bar NULL baz (3 строки)

В функциях array_position и array_positions каждый элемент массива сравнивается с искомым значением по принципу IS NOT DISTINCT FROM.

Функция array_position возвращает NULL, если искомое значение не находится.

Функция array_positions возвращает NULL, только если в качестве массива передаётся NULL; если же в массиве не находится значение, она возвращает пустой массив.

Если для функции string_to_array в качестве разделителя задан NULL, каждый символ входной строки станет отдельным элементом в полученном массиве. Если разделитель пустая строка, строка будет возвращена целиком в массиве из одного элемента. В противном случае входная строка разбивается по вхождениям подстроки, указанной в качестве разделителя.

Если для функции string_to_array параметр замены значения NULL опущен или равен NULL, никакие подстроки во входных данных не будут заменяться на NULL. Если же параметр замены NULL опущен или равен NULL для функции array_to_string, все значения NULL просто пропускаются и никак не представляются в выходной строке.

> **Примечание**
>
> В поведении `string_to_array` по сравнению с PostgreSQL версий до 9.1 произошли два изменения. Во-первых, эта функция возвращает пустой массив (содержащий 0 элементов), а не NULL, когда входная строка имеет нулевую длину. Во-вторых, если в качестве разделителя задан NULL, эта функция разбивает строку по символам, а не просто возвращает NULL, как было раньше.

Вы также можете узнать об агрегатной функции, работающей с массивами, `array_agg` в Разделе 9.20.

9.19. Диапазонные функции и операторы

Диапазонные типы данных рассматриваются в Разделе 8.17.

В Таблице 9.50 показаны операторы, предназначенные для работы с диапазонами.

Таблица 9.50. Диапазонные операторы

Оператор	Описание	Пример	Результат
=	равно	`int4range(1,5) = '[1,4]'::int4range`	t
<>	не равно	`numrange(1.1,2.2) <> numrange(1.1, 2.3)`	t
<	меньше	`int4range(1,10) < int4range(2,3)`	t
>	больше	`int4range(1,10) > int4range(1,5)`	t
<=	меньше или равно	`numrange(1.1,2.2) <= numrange(1.1, 2.2)`	t
>=	больше или равно	`numrange(1.1,2.2) >= numrange(1.1, 2.0)`	t
@>	содержит диапазон	`int4range(2,4) @> int4range(2,3)`	t
@>	содержит элемент	`'[2011-01-01, 2011-03-01)'::tsrange @> '2011-01-10'::timestamp`	t
<@	диапазон содержится в	`int4range(2,4) <@ int4range(1,7)`	t
<@	элемент содержится в	`42 <@ int4range(1, 7)`	f
&&	пересекает (есть общие точки)	`int8range(3,7) && int8range(4,12)`	t
<<	строго слева от	`int8range(1,10) << int8range(100,110)`	t
>>	строго справа от	`int8range(50,60) >> int8range(20,30)`	t

Оператор	Описание	Пример	Результат
`&<`	не простирается правее	`int8range(1,20) &< int8range(18,20)`	`t`
`&>`	не простирается левее	`int8range(7,20) &> int8range(5,10)`	`t`
`-|-`	примыкает к	`numrange(1.1,2.2) -|- numrange(2.2, 3.3)`	`t`
`+`	union	`numrange(5,15) + numrange(10,20)`	`[5,20)`
`*`	пересечение	`int8range(5,15) * int8range(10,20)`	`[10,15)`
`-`	вычитание	`int8range(5,15) - int8range(10,20)`	`[5,10)`

Простые операторы сравнения `<`, `>`, `<=` и `>=` сначала сравнивают нижние границы, и только если они равны, сравнивают верхние. Эти операторы сравнения обычно не очень полезны для диапазонов; основное их предназначение — сделать возможным построение индексов-B-деревьев по диапазонам.

Операторы слева/справа/примыкает всегда возвращают false, если один из диапазонов пуст; то есть, считается, что пустой диапазон находится не слева и не справа от какого-либо другого диапазона.

Операторы сложения и вычитания вызывают ошибку, если получающийся в результате диапазон оказывается состоящим из двух разделённых поддиапазонов, так как его нельзя представить в этом типе данных.

В Таблице 9.51 перечислены функции, предназначенные для работы с диапазонными типами.

Таблица 9.51. Диапазонные функции

Функция	Тип результата	Описание	Пример	Результат
`lower( anyrange)`	тип элемента диапазона	нижняя граница диапазона	`lower(numrange( 1.1, 2.2))`	`1.1`
`upper( anyrange)`	тип элемента диапазона	верхняя граница диапазона	`upper(numrange( 1.1, 2.2))`	`2.2`
`isempty( anyrange)`	`boolean`	диапазон пуст?	`isempty( numrange(1.1, 2.2))`	`false`
`lower_inc ( anyrange)`	`boolean`	нижняя граница включается?	`lower_inc( numrange(1.1, 2.2))`	`true`
`upper_inc ( anyrange)`	`boolean`	верхняя граница включается?	`upper_inc( numrange(1.1, 2.2))`	`false`
`lower_inf ( anyrange)`	`boolean`	нижняя граница равна бесконечности?	`lower_inf('(, )'::daterange)`	`true`
`upper_inf ( anyrange)`	`boolean`	верхняя граница равна бесконечности?	`upper_inf('(, )'::daterange)`	`true`

Функция	Тип результата	Описание	Пример	Результат
range_merge (anyrange, anyrange)	anyrange	наименьший диапазон, включающий оба заданных диапазона	range_merge('[1, 2)'::int4range, '[3, 4)'::int4range)	[1,4)

Функции lower и upper возвращают NULL, если диапазон пуст или указанная граница равна бесконечности. Если же пустой диапазон передаётся функциям lower_inc, upper_inc, lower_inf и upper_inf, все они возвращают false.

9.20. Агрегатные функции

Агрегатные функции получают единственный результат из набора входных значений. Встроенные агрегатные функции общего назначения перечислены в Таблице 9.52, а статистические агрегатные функции — в Таблице 9.53. Встроенные внутригрупповые сортирующие агрегатные функции перечислены в Таблице 9.54, встроенные внутригрупповые гипотезирующие — в Таблице 9.55. Группирующие операторы, тесно связанные с агрегатными функциями, перечислены в Таблице 9.56. Особенности синтаксиса агрегатных функцией разъясняются в Подразделе 4.2.7. За дополнительной вводной информацией обратитесь к Разделу 2.7.

Таблица 9.52. Агрегатные функции общего назначения

Функция	Типы аргумента	Тип результата	Частичный режим	Описание
array_agg(*выражение*)	любой тип не массива	массив элементов с типом аргумента	Нет	входные значения, включая NULL, объединяются в массив
array_agg(*выражение*)	любой тип массива	тот же, что и тип аргумента	Нет	входные массивы собираются в массив большей размерности (они должны иметь одну размерность и не могут быть пустыми или равны NULL)
avg(*выражение*)	smallint, int, bigint, real, double precision, numeric или interval	numeric для любых целочисленных аргументов, double precision для аргументов с плавающей точкой, в противном случае тип данных аргумента	Да	арифметическое среднее для всех входных значений, отличных от NULL
bit_and(*выражение*)	smallint, int, bigint или bit	тот же, что и тип аргумента	Да	побитовое И для всех входных значений, не равных NULL, или NULL, если таких нет
bit_or(*выражение*)	smallint, int, bigint или bit	тот же, что и тип аргумента	Да	побитовое ИЛИ для всех входных

Функция	**Типы аргумента**	**Тип результата**	**Частичный режим**	**Описание**
				значений, не равных NULL, или NULL, если таких нет
`bool_and(` *выражение*`)`	`bool`	`bool`	Да	true, если все входные значения равны true, и false в противном случае
`bool_or(` *выражение*`)`	`bool`	`bool`	Да	true, если хотя бы одно входное значение равно true, и false в противном случае
`count(*)`		`bigint`	Да	количество входных строк
`count(` *выражение*`)`	any	`bigint`	Да	количество входных строк, для которых значение *выражения* не равно NULL
`every(` *выражение*`)`	`bool`	`bool`	Да	синоним `bool_and`
`json_agg(` *выражение*`)`	`any`	`json`	Нет	агрегирует значения, включая NULL, в виде массива JSON
`jsonb_agg(` *выражение*`)`	`any`	`jsonb`	Нет	агрегирует значения, включая NULL, в виде массива JSON
`json_object_agg(` *имя*`,` *значение*`)`	`(any, any)`	`json`	Нет	агрегирует пары имя/значение в виде объекта JSON (NULL допускается в значениях, но не в именах)
`jsonb_object_agg(` *имя*`,` *значение*`)`	`(any, any)`	`jsonb`	Нет	агрегирует пары имя/значение в виде объекта JSON (NULL допускается в значениях, но не в именах)
`max(` *выражение*`)`	любой числовой, строковый, сетевой тип или тип даты/времени, либо массив этих типов	тот же, что и тип аргумента	Да	максимальное значение *выражения* среди всех входных данных, отличных от NULL

Функция	Типы аргумента	Тип результата	Частичный режим	Описание
min(*выражение*)	любой числовой, строковый, сетевой тип или тип даты/времени, либо массив этих типов	тот же, что и тип аргумента	Да	минимальное значение *выражения* среди всех входных данных, отличных от NULL
string_agg(*выражение*, *разделитель*)	(text, text) или (bytea, bytea)	тот же, что и типы аргументов	Нет	входные данные (исключая NULL) складываются в строку через заданный разделитель
sum(*выражение*)	smallint, int, bigint, real, double precision, numeric, interval или money	bigint для аргументов smallint или int, numeric для аргументов bigint, и тип аргумента в остальных случаях	Да	сумма значений *выражения* по всем входным данным, отличным от NULL
xmlagg(*выражение*)	xml	xml	Нет	соединение XML-значений, отличных от NULL (см. также Подраздел 9.14.1.7)

Следует заметить, что за исключением count, все эти функции возвращают NULL, если для них не была выбрана ни одна строка. В частности, функция sum, не получив строк, возвращает NULL, а не 0, как можно было бы ожидать, и array_agg в этом случае возвращает NULL, а не пустой массив. Если необходимо, подставить в результат 0 или пустой массив вместо NULL можно с помощью функции coalesce.

Агрегатные функции, поддерживающие *частичный режим*, являются кандидатами на участие в различных оптимизациях, например, в параллельном агрегировании.

Примечание

Булевы агрегатные функции bool_and и bool_or соответствуют стандартным SQL-агрегатам every и any или some. Что касается any и some, по стандарту их синтаксис допускает некоторую неоднозначность:

```
SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;
```

Здесь ANY можно рассматривать и как объявление подзапроса, и как агрегатную функцию, если этот подзапрос возвращает одну строку с булевым значением. Таким образом, этим агрегатным функциям нельзя было дать стандартные имена.

Примечание

Пользователи с опытом использования других СУБД SQL могут быть недовольны скоростью агрегатной функции count, когда она применяется ко всей таблице. Подобный запрос:

```
SELECT count(*) FROM sometable;
```

потребует затрат в количестве, пропорциональном размеру таблицы: Postgres Pro придётся полностью просканировать либо всю таблицу, либо один из индексов, включающий все её строки.

Агрегатные функции `array_agg`, `json_agg`, `jsonb_agg`, `json_object_agg`, `jsonb_object_agg`, `string_agg` и `xmlagg` так же, как и подобные пользовательские агрегатные функции, выдают разные по содержанию результаты в зависимости от порядка входных значений. По умолчанию порядок не определён, но его можно задать, дополнив вызов агрегатной функции предложением `ORDER BY`, как описано в Подразделе 4.2.7. Обычно нужного результата также можно добиться, передав для агрегирования результат подзапроса с сортировкой. Например:

```
SELECT xmlagg(x) FROM (SELECT x FROM test ORDER BY y DESC) AS tab;
```

Но учтите, что этот подход может не работать, если на внешнем уровне запроса выполняется дополнительная обработка, например, соединение, так как при этом результат подзапроса может быть переупорядочен перед вычислением агрегатной функции.

В Таблице 9.53 перечислены агрегатные функции, обычно применяемые в статистическом анализе. (Они выделены просто для того, чтобы не загромождать список наиболее популярных агрегатных функций.) В их описании под *N* подразумевается число входных строк, для которых входные выражения не равны NULL. Все эти функции возвращают NULL во всех случаях, когда вычисление бессмысленно, например, когда *N* равно 0.

Таблица 9.53. Агрегатные функции для статистических вычислений

Функция	Тип аргумента	Тип результата	Частичный режим	Описание
`corr(` *Y*`,` *X*`)`	`double precision`	`double precision`	Да	коэффициент корреляции
`covar_pop(` *Y*`,` *X*`)`	`double precision`	`double precision`	Да	ковариация совокупности
`covar_samp(` *Y*`,` *X*`)`	`double precision`	`double precision`	Да	ковариация выборки
`regr_avgx(` *Y*`,` *X*`)`	`double precision`	`double precision`	Да	среднее независимой переменной (`sum(` *X*`) /` *N*`)`
`regr_avgy(` *Y*`,` *X*`)`	`double precision`	`double precision`	Да	среднее зависимой переменной (`sum(` *Y*`) /` *N*`)`
`regr_count(` *Y*`,` *X*`)`	`double precision`	`bigint`	Да	число входных строк, в которых оба выражения не NULL
`regr_intercept(` *Y*`,` *X*`)`	`double precision`	`double precision`	Да	пересечение с осью OY линии, полученной методом наименьших квадратов по данным (*X*, *Y*)
`regr_r2(` *Y*`,` *X*`)`	`double precision`	`double precision`	Да	квадрат коэффициента корреляции

Функция	**Тип аргумента**	**Тип результата**	**Частичный режим**	**Описание**
regr_slope(*Y*, *X*)	double precision	double precision	Да	наклон линии, полученной методом наименьших квадратов по данным (*X*, *Y*)
regr_sxx(*Y*, *X*)	double precision	double precision	Да	sum(*X*^2) - sum(*X*)^2/*N* («сумма квадратов» независимой переменной)
regr_sxy(*Y*, *X*)	double precision	double precision	Да	sum(*X***Y*) - sum(*X*) * sum(*Y*)/*N* («сумма произведений» независимых и зависимых переменных)
regr_syy(*Y*, *X*)	double precision	double precision	Да	sum(*Y*^2) - sum(*Y*)^2/*N* («сумма квадратов» зависимой переменной)
stddev(*выражение*)	smallint, int, bigint, real, double precision или numeric	double precision для аргументов с плавающей точкой, numeric для остальных	Да	сохранившийся синоним stddev_samp
stddev_pop(*выражение*)	smallint, int, bigint, real, double precision или numeric	double precision для аргументов с плавающей точкой, numeric для остальных	Да	стандартное отклонение по генеральной совокупности входных значений
stddev_samp(*выражение*)	smallint, int, bigint, real, double precision или numeric	double precision для аргументов с плавающей точкой, numeric для остальных	Да	стандартное отклонение по выборке входных значений
variance(*выражение*)	smallint, int, bigint, real, double precision или numeric	double precision для аргументов с плавающей точкой, numeric для остальных	Да	сохранившийся синоним var_samp
var_pop (*выражение*)	smallint, int, bigint, real, double precision или numeric	double precision для аргументов с плавающей точкой, numeric для остальных	Да	дисперсия для генеральной совокупности входных значений (квадрат стандартного отклонения)

Функция	Тип аргумента	Тип результата	Частичный режим	Описание
var_samp (*выражение*)	smallint, int, bigint, real, double precision или numeric	double precision для аргументов с плавающей точкой, numeric для остальных	Да	дисперсия по выборке для входных значений (квадрат отклонения по выборке)

В Таблице 9.54 показаны некоторые агрегатные функции, использующие синтаксис *сортирующих агрегатных функций*. Иногда такие функции функциями называют функциями «обратного распределения».

Таблица 9.54. Сортирующие агрегатные функции

Функция	Тип непосредственного аргумента	Тип сортированного аргумента	Тип результата	Частичный режим	Описание
mode() WITHIN GROUP (ORDER BY *выражение_сортировки*)		любой сортируемый тип	тот же, что у выражения сортировки	Нет	возвращает значение, наиболее часто встречающееся во входных данных (если одинаково часто встречаются несколько значений, произвольно выбирается первое из них)
percentile_cont(*дробь*) WITHIN GROUP (ORDER BY *выражение_сортировки*)	double precision	double precision или interval	тот же, что у выражения сортировки	Нет	непрерывный процентиль: возвращает значение, соответствующее заданной дроби по порядку, интерполируя соседние входные значения, если необходимо
percentile_cont(*дроби*) WITHIN GROUP (ORDER BY *выражение_сортировки*)	double precision[]	double precision или interval	массив типа выражения сортировки	Нет	множественный непрерывный процентиль: возвращает массив результатов, соответствующих форме параметра

Функция	Тип непосредственного аргумента	Тип агрегированного аргумента	Тип результата	Частичный режим	Описание
					дроби (для каждого элемента не NULL подставляется значение, соответствующее данному процентилю)
percentile_disc(*дробь*) WITHIN GROUP (ORDER BY *выражение_сортировки*)	double precision	любой сортируемый тип	тот же, что у выражения сортировки	Нет	дискретный процентиль: возвращает первое значение из входных данных, позиция которого по порядку равна или превосходит указанную дробь
percentile_disc(*дроби*) WITHIN GROUP (ORDER BY *выражение_сортировки*)	double precision[]	любой сортируемый тип	массив типа выражения сортировки	Нет	множественный дискретный процентиль: возвращает массив результатов, соответствующих форме параметра *дроби* (для каждого элемента не NULL подставляется входное значение, соответствующее данному процентилю)

Все агрегатные функции, перечисленные в Таблице 9.54, игнорируют значения NULL при сортировке данных. Для функций, принимающих параметр *дробь*, значение этого параметра должно быть от 0 до 1; в противном случае возникает ошибка. Однако если в этом параметре передаётся NULL, эти функции просто выдают NULL.

Все агрегатные функции, перечисленные в Таблице 9.55, связаны с одноимёнными оконными функциями, определёнными в Разделе 9.21. В каждом случае их результат — значение, которое вернула бы связанная оконная функция для «гипотетической» строки, полученной из *аргументов*, если бы такая строка была добавлена в сортированную группу строк, которую образуют *сортированные_аргументы*.

Таблица 9.55. Гипотезирующие агрегатные функции

Функция	Тип непосредственного аргумента	Тип сортированного аргумента	Тип результата	Частичный режим	Описание
rank(*аргументы*) WITHIN GROUP (ORDER BY *сортированные_аргументы*)	VARIADIC "any"	VARIADIC "any"	bigint	Нет	ранг гипотетической строки, с пропусками повторяющихся строк
dense_rank(*аргументы*) WITHIN GROUP (ORDER BY *сортированные_аргументы*)	VARIADIC "any"	VARIADIC "any"	bigint	Нет	ранг гипотетической строки, без пропусков
percent_rank(*аргументы*) WITHIN GROUP (ORDER BY *сортированные_аргументы*)	VARIADIC "any"	VARIADIC "any"	double precision	Нет	относительный ранг гипотетической строки, от 0 до 1
cume_dist(*аргументы*) WITHIN GROUP (ORDER BY *сортированные_аргументы*)	VARIADIC "any"	VARIADIC "any"	double precision	Нет	относительный ранг гипотетической строки, от 1/*N* до 1

Для всех этих гипотезирующих агрегатных функций непосредственные *аргументы* должны соответствовать (по количеству и типу) *сортированным_аргументам*. В отличие от встроенных агрегатных функций, они не являются строгими, то есть не отбрасывают входные строки, содержащие NULL. Значения NULL сортируются согласно правилу, указанному в предложении ORDER BY.

Таблица 9.56. Операции группировки

Функция	Тип результата	Описание
GROUPING(*аргументы...*)	integer	Целочисленная битовая маска, показывающая, какие аргументы не вошли в текущий набор группирования

Операции группировки применяются в сочетании с наборами группирования (см. Подраздел 7.2.4) для различения результирующих строк. Аргументы операции GROUPING на самом деле не вычисляются, но они должны в точности соответствовать выражениям, заданным в предложении GROUP BY на их уровне запроса. Биты назначаются справа налево (правый аргумент отражается в младшем бите); бит равен 0, если соответствующее выражение вошло в критерий группировки набора группирования, для которого сформирована строка результата, или 1 в противном случае. Например:

```
=> SELECT * FROM items_sold;
 make  | model | sales
```

```
-------+-------+-------
 Foo   | GT    |  10
 Foo   | Tour  |  20
 Bar   | City  |  15
 Bar   | Sport |  5
(4 rows)

=> SELECT make, model, GROUPING(make,model), sum(sales) FROM items_sold GROUP BY
 ROLLUP(make,model);
 make  | model | grouping | sum
-------+-------+----------+-----
 Foo   | GT    |        0 | 10
 Foo   | Tour  |        0 | 20
 Bar   | City  |        0 | 15
 Bar   | Sport |        0 | 5
 Foo   |       |        1 | 30
 Bar   |       |        1 | 20
       |       |        3 | 50
(7 rows)
```

9.21. Оконные функции

Оконные функции дают возможность выполнять вычисления с набором строк, каким-либо образом связанным с текущей строкой запроса. Вводную информацию об этом можно получить в Разделе 3.5, а подробнее узнать о синтаксисе можно в Подразделе 4.2.8.

Встроенные оконные функции перечислены в Таблице 9.57. Заметьте, что эти функции *должны* вызываться именно как оконные, т. е. при вызове необходимо использовать предложение `OVER`.

В дополнение к этим функциям в качестве оконных можно использовать любые встроенные или пользовательские универсальные или статистические агрегатные функции (но не сортирующие и не гипотезирующие); список встроенных агрегатных функций приведён в Разделе 9.20. Агрегатные функции работают как оконные, только когда за их вызовом следует предложение `OVER`; в противном случае они работают как обычные, не оконные функции и выдают для всего набора единственную строку.

Таблица 9.57. Оконные функции общего назначения

Функция	Тип результата	Описание
`row_number()`	`bigint`	номер текущей строки в её разделе, начиная с 1
`rank()`	`bigint`	ранг текущей строки с пропусками; то же, что и `row_number` для первой родственной ей строки
`dense_rank()`	`bigint`	ранг текущей строки без пропусков; эта функция считает группы родственных строк
`percent_rank()`	`double precision`	относительный ранг текущей строки: (`rank` - 1) / (общее число строк раздела - 1)
`cume_dist()`	`double precision`	кумулятивное распределение: (число строк раздела, предшествующих или родственных текущей строке) / общее число строк раздела

Функция	**Тип результата**	**Описание**
ntile(*число_групп* integer)	integer	ранжирование по целым числам от 1 до значения аргумента так, чтобы размеры групп были максимально близки
lag(*значение* anyelement [, *смещение* integer [, *по_умолчанию* anyelement]])	тип аргумента *значение*	возвращает *значение* для строки, положение которой задаётся *смещением* от текущей строки к началу раздела; если такой строки нет, возвращается значение *по_умолчанию* (оно должно иметь тот же тип, что и *значение*). Оба параметра *смещение* и *по_умолчанию* вычисляются для текущей строки. Если они не указываются, то *смещение* считается равным 1, а *по_умолчанию* — NULL
lead(*значение* anyelement [, *смещение* integer [, *по_умолчанию* anyelement]])	тип аргумента *значение*	возвращает *значение* для строки, положение которой задаётся *смещением* от текущей строки к концу раздела; если такой строки нет, возвращается значение *по_умолчанию* (оно должно иметь тот же тип, что и *значение*). Оба параметра *смещение* и *по_умолчанию* вычисляются для текущей строки. Если они не указываются, то *смещение* считается равным 1, а *по_умолчанию* — NULL
first_value(*значение* any)	тип аргумента *значение*	возвращает *значение*, вычисленное для первой строки в рамке окна
last_value(*значение* any)	тип аргумента *значение*	возвращает *значение*, вычисленное для последней строки в рамке окна
nth_value(*значение* any, *n* integer)	тип аргумента *значение*	возвращает *значение*, вычисленное в *n-ой* строке в рамке окна (считая с 1), или NULL, если такой строки нет

Результат всех функций, перечисленных в Таблице 9.57, зависит от порядка сортировки, заданного предложением ORDER BY в определении соответствующего окна. Строки, которые являются одинаковыми при рассмотрении только столбцов ORDER BY, считаются *родственными*. Четыре функции, вычисляющие ранг (включая cume_dist), реализованы так, что их результат будет одинаковым для всех родственных строк.

Заметьте, что функции first_value, last_value и nth_value рассматривают только строки в «рамке окна», которая по умолчанию содержит строки от начала раздела до последней родственной строки для текущей. Поэтому результаты last_value и иногда nth_value могут быть

не очень полезны. В таких случаях можно переопределить рамку, добавив в предложение OVER подходящее указание (RANGE или ROWS). Подробнее эти указания описаны в Подразделе 4.2.8.

Когда в качестве оконной функции используется агрегатная, она обрабатывает строки в рамке текущей строки. Агрегатная функция с ORDER BY и определением рамки окна по умолчанию будет вычисляться как «бегущая сумма», что может не соответствовать желаемому результату. Чтобы агрегатная функция работала со всем разделом, следует опустить ORDER BY или использовать ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING. Используя другие указания в определении рамки, можно получить и другие эффекты.

> **Примечание**
>
> В стандарте SQL определены параметры RESPECT NULLS или IGNORE NULLS для функций lead, lag, first_value, last_value и nth_value. В Postgres Pro такие параметры не реализованы: эти функции ведут себя так, как положено в стандарте по умолчанию (или с подразумеваемым параметром RESPECT NULLS). Также функция nth_value не поддерживает предусмотренные стандартом параметры FROM FIRST и FROM LAST: реализовано только поведение по умолчанию (с подразумеваемым параметром FROM FIRST). (Получить эффект параметра FROM LAST можно, изменив порядок ORDER BY на обратный.)

Функция cume_dist вычисляет процент строк раздела, которые меньше или равны текущей строке или родственным ей строкам, тогда как percent_rank вычисляет процент строк раздела, которые меньше текущей строки, в предположении, что текущая строка не относится к разделу.

9.22. Выражения подзапросов

В этом разделе описаны выражения подзапросов, которые реализованы в Postgres Pro в соответствии со стандартом SQL. Все рассмотренные здесь формы выражений возвращает булевы значения (true/false).

9.22.1. EXISTS

```
EXISTS (подзапрос)
```

Аргументом EXISTS является обычный оператор SELECT, т. е. *подзапрос*. Выполнив запрос, система проверяет, возвращает ли он строки в результате. Если он возвращает минимум одну строку, результатом EXISTS будет «true», а если не возвращает ни одной — «false».

Подзапрос может обращаться к переменным внешнего запроса, которые в рамках одного вычисления подзапроса считаются константами.

Вообще говоря, подзапрос может выполняться не полностью, а завершаться, как только будет возвращена хотя бы одна строка. Поэтому в подзапросах следует избегать побочных эффектов (например, обращений к генераторам последовательностей); проявление побочного эффекта может быть непредсказуемым.

Так как результат этого выражения зависит только от того, возвращаются строки или нет, но не от их содержимого, список выходных значений подзапроса обычно не имеет значения. Как следствие, широко распространена практика, когда проверки EXISTS записываются в форме EXISTS(SELECT 1 WHERE ...). Однако из этого правила есть и исключения, например с подзапросами с предложением INTERSECT.

Этот простой пример похож на внутреннее соединение по столбцу col2, но он выдаёт максимум одну строку для каждой строки в tab1, даже если в tab2 ей соответствуют несколько строк:

```
SELECT col1
FROM tab1
WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

9.22.2. IN

выражение IN (*подзапрос*)

В правой стороне этого выражения в скобках задаётся подзапрос, который должен возвращать ровно один столбец. Вычисленное значение левого выражения сравнивается со значениями во всех строках, возвращённых подзапросом. Результатом всего выражения IN будет «true», если строка с таким значением находится, и «false» в противном случае (в том числе, когда подзапрос вообще не возвращает строк).

Заметьте, что если результатом выражения слева оказывается NULL или равных значений справа не находится, а хотя бы одно из значений справа равно NULL, конструкция IN возвращает NULL, а не false. Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

Так же, как и с EXISTS, здесь не следует рассчитывать на то, что подзапрос будет всегда выполняться полностью.

конструктор_строки IN (*подзапрос*)

В левой части этой формы IN записывается конструктор строки (подробнее они рассматриваются в Подразделе 4.2.13). Справа в скобках записывается подзапрос, который должен вернуть ровно столько столбцов, сколько содержит строка в выражении слева. Вычисленные значения левого выражения сравниваются построчно со значениями во всех строках, возвращённых подзапросом. Результатом всего выражения IN будет «true», если строка с такими значениями находится, и «false» в противном случае (в том числе, когда подзапрос вообще не возвращает строк).

Как обычно, значения NULL в строках обрабатываются при этом по принятым в SQL правилам сравнения. Две строки считаются равными, если все их соответствующие элементы не равны NULL, но равны между собой; неравными они считаются, когда в них находятся элементы, не равные NULL, и не равные друг другу; в противном случае результат сравнения строк не определён (равен NULL). Если в результатах сравнения строк нет ни одного положительного, но есть хотя бы один NULL, результатом IN будет NULL.

9.22.3. NOT IN

выражение NOT IN (*подзапрос*)

Справа в скобках записывается подзапрос, который должен возвращать ровно один столбец. Вычисленное значение левого выражения сравнивается со значением во всех строках, возвращённых подзапросом. Результатом всего выражения NOT IN будет «true», если находятся только несовпадающие строки (в том числе, когда подзапрос вообще не возвращает строк). Если же находится хотя бы одна подходящая строка, результатом будет «false».

Заметьте, что если результатом выражения слева оказывается NULL или равных значений справа не находится, а хотя бы одно из значений справа равно NULL, конструкция NOT IN возвращает NULL, а не true. Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

Так же, как и с EXISTS, здесь не следует рассчитывать на то, что подзапрос будет всегда выполняться полностью.

конструктор_строки NOT IN (*подзапрос*)

В левой части этой формы NOT IN записывается конструктор строки (подробнее они описываются в Подразделе 4.2.13). Справа в скобках записывается подзапрос, который должен вернуть ровно столько столбцов, сколько содержит строка в выражении слева. Вычисленные значения левого выражения сравниваются построчно со значениями во всех строках, возвращённых подзапросом. Результатом всего выражения NOT IN будет «true», если равных строк не найдётся (в том числе, и когда подзапрос не возвращает строк), и «false», если такие строки есть.

Как обычно, значения NULL в строках обрабатываются при этом по принятым в SQL правилам сравнения. Две строки считаются равными, если все их соответствующие элементы не равны NULL, но равны между собой; неравными они считаются, когда в них находятся элементы, не

равные NULL, и не равные друг другу; в противном случае результат сравнения строк не определён (равен NULL). Если в результатах сравнения строк нет ни одного положительного, но есть хотя бы один NULL, результатом NOT IN будет NULL.

9.22.4. ANY/SOME

выражение оператор ANY (*подзапрос*)
выражение оператор SOME (*подзапрос*)

В правой части конструкции в скобках записывается подзапрос, который должен возвращать ровно один столбец. Вычисленное значение левого выражения сравнивается со значением в каждой строке результата подзапроса с помощью заданного *оператора* условия, который должен выдавать логическое значение. Результатом ANY будет «true», если хотя бы для одной строки условие истинно, и «false» в противном случае (в том числе, и когда подзапрос не возвращает строк).

Ключевое слово SOME является синонимом ANY. Конструкцию IN можно также записать как = ANY.

Заметьте, что если условие не выполняется ни для одной из строк, а хотя бы для одной строки условный оператор выдаёт NULL, конструкция ANY возвращает NULL, а не false. Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

Так же, как и с EXISTS, здесь не следует рассчитывать на то, что подзапрос будет всегда выполняться полностью.

конструктор_строки оператор ANY (*подзапрос*)
конструктор_строки оператор SOME (*подзапрос*)

В левой части этой формы ANY записывается конструктор строки (подробнее они описываются в Подразделе 4.2.13). Справа в скобках записывается подзапрос, который должен возвращать ровно столько столбцов, сколько содержит строка в выражении слева. Вычисленные значения левого выражения сравниваются построчно со значениями во всех строках, возвращённых подзапросом, с применением заданного *оператора*. Результатом всего выражения ANY будет «true», если для какой-либо из строк подзапроса результатом сравнения будет true, или «false», если для всех строк результатом сравнения оказывается false (в том числе, и когда подзапрос не возвращает строк). Результатом выражения будет NULL, если ни для одной из строк подзапроса результат сравнения не равен true, а минимум для одной равен NULL.

Подробнее логика сравнения конструкторов строк описана в Подразделе 9.23.5.

9.22.5. ALL

выражение оператор ALL (*подзапрос*)

В правой части конструкции в скобках записывается подзапрос, который должен возвращать ровно один столбец. Вычисленное значение левого выражения сравнивается со значением в каждой строке результата подзапроса с помощью заданного *оператора* условия, который должен выдавать логическое значение. Результатом ALL будет «true», если условие истинно для всех строк (и когда подзапрос не возвращает строк), или «false», если находятся строки, для которых оно ложно. Результатом выражения будет NULL, если ни для одной из строк подзапроса результат сравнения не равен true, а минимум для одной равен NULL.

Конструкция NOT IN равнозначна <> ALL.

Так же, как и с EXISTS, здесь не следует рассчитывать на то, что подзапрос будет всегда выполняться полностью.

конструктор_строки оператор ALL (*подзапрос*)

В левой части этой формы ALL записывается конструктор строки (подробнее они описываются в Подразделе 4.2.13). Справа в скобках записывается подзапрос, который должен возвращать ровно столько столбцов, сколько содержит строка в выражении слева. Вычисленные значения левого выражения сравниваются построчно со значениями во всех строках, возвращённых подзапросом, с применением заданного *оператора*. Результатом всего выражения ALL будет «true», если для всех

строк подзапроса результатом сравнения будет true (или если подзапрос не возвращает строк), либо «false», если результат сравнения равен false для любой из строк подзапроса. Результатом выражения будет NULL, если ни для одной из строк подзапроса результат сравнения не равен true, а минимум для одной равен NULL.

Подробнее логика сравнения конструкторов строк описана в Подразделе 9.23.5.

9.22.6. Сравнение единичных строк

конструктор_строки оператор (*подзапрос*)

В левой части конструкции записывается конструктор строки (подробнее они описываются в Подразделе 4.2.13). Справа в скобках записывается подзапрос, который должен возвращать ровно столько столбцов, сколько содержит строка в выражении слева. Более того, подзапрос может вернуть максимум одну строку. (Если он не вернёт строк, результатом будет NULL.) Конструкция возвращает результат сравнения строки слева с этой одной строкой результата подзапроса.

Подробнее логика сравнения конструкторов строк описана в Подразделе 9.23.5.

9.23. Сравнение табличных строк и массивов

В этом разделе описываются несколько специальных конструкций, позволяющих сравнивать группы значений. Синтаксис этих конструкций связан с формами выражений с подзапросами, описанными в предыдущем разделе, а отличаются они отсутствием подзапросов. Конструкции, в которых в качестве подвыражений используются массивы, являются расширениями Postgres Pro; все остальные формы соответствуют стандарту SQL. Все описанные здесь выражения возвращают логические значения (true/false).

9.23.1. IN

выражение IN (*значение* [, ...])

Справа в скобках записывается список скалярных выражений. Результатом будет «true», если значение левого выражения равняется одному из значений выражений в правой части. Эту конструкцию можно считать краткой записью условия

```
выражение = значение1
OR
выражение = значение2
OR
...
```

Заметьте, что если результатом выражения слева оказывается NULL или равных значений справа не находится, а хотя бы одно из значений справа равно NULL, конструкция IN возвращает NULL, а не false. Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

9.23.2. NOT IN

выражение NOT IN (*значение* [, ...])

Справа в скобках записывается список скалярных выражений. Результатом будет «true», если значение левого выражения не равно ни одному из значений выражений в правой части. Эту конструкцию можно считать краткой записью условия

```
выражение <> значение1
AND
выражение <> значение2
AND
...
```

Заметьте, что если результатом выражения слева оказывается NULL или равных значений справа не находится, а хотя бы одно из значений справа равно NULL, конструкция NOT IN возвращает NULL, а не true, как можно было бы наивно полагать. Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

> **Подсказка**
>
> Выражения `x NOT IN y` и `NOT (x IN y)` полностью равнозначны. Учитывая, что значения NULL могут ввести в заблуждение начинающих скорее в конструкции `NOT IN`, чем в `IN`, лучше формулировать условия так, чтобы в них было как можно меньше отрицаний.

9.23.3. `ANY`/`SOME` (с массивом)

выражение оператор `ANY (`*выражение массива*`)`
выражение оператор `SOME (`*выражение массива*`)`

Справа в скобках записывается выражение, результатом которого является массив. Вычисленное значение левого выражения сравнивается с каждым элементом этого массива с применением заданного *оператора* условия, который должен выдавать логическое значение. Результатом `ANY` будет «true», если для какого-либо элемента условие истинно, и «false» в противном случае (в том числе, и когда массив оказывается пустым).

Если значением массива оказывается NULL, результатом `ANY` также будет NULL. Если NULL получен в левой части, результатом `ANY` обычно тоже будет NULL (хотя оператор нестрогого сравнения может выдать другой результат). Кроме того, если массив в правой части содержит элементы NULL и ни с одним из элементов условие не выполняется, результатом `ANY` будет NULL, а не false (опять же, если используется оператор строгого сравнения). Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

Ключевое слово `SOME` является синонимом `ANY`.

9.23.4. `ALL` (с массивом)

выражение оператор `ALL (`*выражение массива*`)`

Справа в скобках записывается выражение, результатом которого является массив. Вычисленное значение левого выражения сравнивается с каждым элементом этого массива с применением заданного *оператора* условия, который должен выдавать логическое значение. Результатом `ALL` будет «true», если для всех элементов условие истинно (или массив не содержит элементов), и «false», если находятся строки, для которых оно ложно.

Если значением массива оказывается NULL, результатом `ALL` также будет NULL. Если NULL получен в левой части, результатом `ALL` обычно тоже будет NULL (хотя оператор нестрогого сравнения может выдать другой результат). Кроме того, если массив в правой части содержит элементы NULL и при этом нет элементов, с которыми условие не выполняется, результатом `ALL` будет NULL, а не true (опять же, если используется оператор строгого сравнения). Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

9.23.5. Сравнение конструкторов строк

конструктор_строки оператор конструктор_строки

С обеих сторон представлены конструкторы строк (они описываются в Подразделе 4.2.13). При этом данные строки должны содержать одинаковое число полей. После вычисления каждой стороны они сравниваются по строкам. Сравнения конструкторов строк возможны с *оператором* `=`, `<>`, `<`, `<=`, `>` или `>=`. Каждый элемент строки должен иметь тип, для которого определён класс операторов B-дерева; в противном случае при попытке сравнения может возникнуть ошибка.

> **Примечание**
>
> Ошибок, связанных с числом или типов элементов, не должно быть, если сравнение выполняется с ранее полученными столбцами.

Сравнения = и <> несколько отличаются от других. С этими операторами две строки считаются равными, если все их соответствующие поля не равны NULL и равны между собой, и неравными, если какие-либо соответствующие их поля не NULL и не равны между собой. В противном случае результатом сравнения будет неопределённость (NULL).

С операторами <, <=, > и >= элементы строк сравниваются слева направо до тех пор, пока не будет найдена пара неравных элементов или значений NULL. Если любым из элементов пары оказывается NULL, результатом сравнения будет неопределённость (NULL), в противном случае результат всего выражения определяется результатом сравнения этих двух элементов. Например, результатом `ROW(1,2,NULL) < ROW(1,3,0)` будет true, а не NULL, так как третья пара элементов не принимается в рассмотрение.

> **Примечание**
>
> До версии 8.2 PostgreSQL обрабатывал условия <, <=, > и >= не так, как это описано в стандарте SQL. Сравнение `ROW(a,b) < ROW(c,d)` выполнялось как `a < c AND b < d`, тогда как по стандарту должно быть `a < c OR (a = c AND b < d)`.

конструктор_строки `IS DISTINCT FROM` *конструктор_строки*

Эта конструкция похожа на сравнение строк с оператором <>, но со значениями NULL она выдаёт не NULL. Любое значение NULL для неё считается неравным (отличным от) любому значению не NULL, а два NULL считаются равными (не различными). Таким образом, результатом такого выражения будет true или false, но не NULL.

конструктор_строки `IS NOT DISTINCT FROM` *конструктор_строки*

Эта конструкция похожа на сравнение строк с оператором =, но со значениями NULL она выдаёт не NULL. Любое значение NULL для неё считается неравным (отличным от) любому значению не NULL, а два NULL считаются равными (не различными). Таким образом, результатом такого выражения всегда будет true или false, но не NULL.

9.23.6. Сравнение составных типов

запись оператор запись

Стандарт SQL требует, чтобы при сравнении строк возвращался NULL, если результат зависит от сравнения двух значений NULL или значения NULL и не NULL. Postgres Pro выполняет это требование только при сравнении строк, созданных конструкторами (как описано в Подразделе 9.23.5), или строки, созданной конструктором, со строкой результата подзапроса (как было описано в Разделе 9.22). В других контекстах при сравнении полей составных типов два значения NULL считаются равными, а любое значение не NULL полагается меньшим NULL. Это отклонение от правила необходимо для полноценной реализации сортировки и индексирования составных типов.

После вычисления каждой стороны они сравниваются по строкам. Сравнения составных типов возможны с *оператором* =, <>, <, <=, > или >=, либо другим подобным. (Точнее, оператором сравнения строк может быть любой оператор, входящий в класс операторов B-дерева, либо обратный к оператору =, входящему в класс операторов B-дерева.) По умолчанию вышеперечисленные операторы действуют так же, как выражение `IS [ NOT ] DISTINCT FROM` для конструкторов строк (см. Подраздел 9.23.5).

Для поддержки сравнения строк с элементами, для которых не определён класс операторов B-дерева по умолчанию, введены следующие операторы: *=, *<>, *<, *<=, *> и *>=. Эти операторы сравнивают внутреннее двоичное представление двух строк. Учтите, что две строки могут иметь различное двоичное представление, даже когда при сравнении оператором равенства считаются равными. Порядок строк с такими операторами детерминирован, но не несёт смысловой нагрузки. Данные операторы применяются внутри системы для материализованных представлений и могут быть полезны для других специальных целей (например, репликации), но, вообще говоря, не предназначены для использования в обычных запросах.

9.24. Функции, возвращающие множества

В этом разделе описаны функции, которые могут возвращать не одну, а множество строк. Чаще всего из их числа используются функции, генерирующие ряды значений, которые перечислены в Таблице 9.58 и Таблице 9.59. Другие, более специализированные функции множеств описаны в других разделах этой документации. Варианты комбинирования нескольких функций, возвращающих множества строк, описаны в Подразделе 7.2.1.4.

Таблица 9.58. Функции, генерирующие ряды значений

Функция	Тип аргумента	Тип результата	Описание
`generate_series( start, stop)`	`int`, `bigint` или `numeric`	`setof int`, `setof bigint` или `setof numeric` (определяется типом аргумента)	Выдаёт ряд целых чисел от *start* до *stop* с шагом 1
`generate_series( start, stop, step)`	`int`, `bigint` или `numeric`	`setof int`, `setof bigint` или `setof numeric` (определяется типом аргумента)	Выдаёт ряд значений от *start* до *stop* с заданным шагом (*step*)
`generate_series( start, stop, step interval)`	`timestamp` или `timestamp with time zone`	`setof timestamp` или `setof timestamp with time zone` (определяется типом аргумента)	Выдаёт ряд значений от *start* до *stop* с заданным шагом (*step*)

Если заданный шаг (*step*) положительный, а *start* оказывается больше *stop*, эти функции возвращают 0 строк. Тот же результат будет, если *step* меньше 0, а *start* меньше *stop*, или если какой-либо аргумент равен `NULL`. Если же *step* будет равен 0, произойдёт ошибка. Несколько примеров:

```
SELECT * FROM generate_series(2,4);
 generate_series
-----------------
               2
               3
               4
(3 rows)

SELECT * FROM generate_series(5,1,-2);
 generate_series
-----------------
               5
               3
               1
(3 rows)

SELECT * FROM generate_series(4,3);
 generate_series
-----------------
(0 rows)

SELECT generate_series(1.1, 4, 1.3);
 generate_series
-----------------
             1.1
             2.4
             3.7
```

```
(3 rows)

-- этот пример задействует оператор прибавления к дате целого числа
SELECT current_date + s.a AS dates FROM generate_series(0,14,7) AS s(a);
   dates
------------
 2004-02-05
 2004-02-12
 2004-02-19
(3 rows)

SELECT * FROM generate_series('2008-03-01 00:00'::timestamp,
                              '2008-03-04 12:00', '10 hours');
   generate_series
---------------------
 2008-03-01 00:00:00
 2008-03-01 10:00:00
 2008-03-01 20:00:00
 2008-03-02 06:00:00
 2008-03-02 16:00:00
 2008-03-03 02:00:00
 2008-03-03 12:00:00
 2008-03-03 22:00:00
 2008-03-04 08:00:00
(9 rows)
```

Таблица 9.59. Функции, генерирующие индексы массивов

Функция	Тип результата	Описание
generate_subscripts(*array anyarray*, *dim int*)	setof int	Выдаёт ряд значений для использования в качестве индекса данного массива.
generate_subscripts(*array anyarray*, *dim int*, *reverse boolean*)	setof int	Выдаёт ряд значений для использования в качестве индекса данного массива. Если параметр *reverse* равен true, значения выдаются от большего к меньшему.

Функция generate_subscripts позволяет упростить получение всего набора индексов для указанной размерности заданного массива. Она выдаёт 0 строк, если в массиве нет указанной размерности или сам массив равен NULL (хотя для элементов, равных NULL, индексы будут выданы, как и для любых других). Взгляните на следующие примеры:

```
-- простой пример использования
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;
 s
---
 1
 2
 3
 4
(4 rows)

-- для показанного массива получение индекса и обращение
-- к элементу по индексу выполняется с помощью подзапроса
SELECT * FROM arrays;
```

```
         a
--------------------
 {-1,-2}
 {100,200,300}
(2 rows)

SELECT a AS array, s AS subscript, a[s] AS value
FROM (SELECT generate_subscripts(a, 1) AS s, a FROM arrays) foo;
     array     | subscript | value
---------------+-----------+-------
 {-1,-2}       |         1 |    -1
 {-1,-2}       |         2 |    -2
 {100,200,300} |         1 |   100
 {100,200,300} |         2 |   200
 {100,200,300} |         3 |   300
(5 rows)

-- разворачивание двумерного массива
CREATE OR REPLACE FUNCTION unnest2(anyarray)
RETURNS SETOF anyelement AS $$
select $1[i][j]
   from generate_subscripts($1,1) g1(i),
        generate_subscripts($1,2) g2(j);
$$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);
 unnest2
---------
       1
       2
       3
       4
(4 rows)
```

Когда после функции в предложении `FROM` добавляется `WITH ORDINALITY`, в выходные данные добавляется столбец типа `bigint`, числа в котором начинаются с 1 и увеличиваются на 1 для каждой строки, выданной функцией. В первую очередь это полезно для функций, возвращающих множества, например, `unnest()`.

```
-- функция, возвращающая множество, с нумерацией
SELECT * FROM pg_ls_dir('.') WITH ORDINALITY AS t(ls,n);
       ls        | n
-----------------+----
 pg_serial       |  1
 pg_twophase     |  2
 postmaster.opts |  3
 pg_notify       |  4
 postgresql.conf |  5
 pg_tblspc       |  6
 logfile         |  7
 base            |  8
 postmaster.pid  |  9
 pg_ident.conf   | 10
 global          | 11
 pg_xact         | 12
 pg_snapshots    | 13
 pg_multixact    | 14
 PG_VERSION      | 15
```

```
 pg_wal           | 16
 pg_hba.conf      | 17
 pg_stat_tmp      | 18
 pg_subtrans      | 19
(19 строк)
```

9.25. Системные информационные функции

В Таблице 9.60 перечислен ряд функций, предназначенных для получения информации о текущем сеансе и системе.

В дополнение к перечисленным здесь функциям существуют также функции, связанные с подсистемой статистики, которые тоже предоставляют системную информацию. Подробнее они рассматриваются в Подразделе 27.2.2.

Таблица 9.60. Функции получения информации о сеансе

Имя	Тип результата	Описание
`current_catalog`	`name`	имя текущей базы данных (в стандарте SQL она называется «каталогом»)
`current_database()`	`name`	имя текущей базы данных
`current_query()`	`text`	текст запроса, выполняемого в данный момент, в том виде, в каком его передал клиент (может состоять из нескольких операторов)
`current_role`	`name`	синоним `current_user`
`current_schema [()]`	`name`	имя текущей схемы
`current_schemas( boolean)`	`name[]`	имена схем в пути поиска, возможно включая схемы, добавляемые в него неявно
`current_user`	`name`	имя пользователя в текущем контексте выполнения
`inet_client_addr()`	`inet`	адрес удалённой стороны соединения
`inet_client_port()`	`int`	порт удалённой стороны соединения
`inet_server_addr()`	`inet`	адрес локальной стороны соединения
`inet_server_port()`	`int`	порт локальной стороны соединения
`pg_backend_pid()`	`int`	код серверного процесса, обслуживающего текущий сеанс
`pg_blocking_pids( int)`	`int[]`	идентификаторы процессов, не дающих серверному процессу с определённым ID получить блокировку
`pg_conf_load_time()`	`timestamp with time zone`	время загрузки конфигурации
`pg_current_logfile( [ text])`	`text`	имя файла главного журнала или журнала в заданном формате, который в настоящее

Имя	Тип результата	Описание
		время используется сборщиком сообщений
pg_my_temp_schema()	oid	OID временной схемы этого сеанса или 0, если её нет
pg_is_other_temp_schema(oid)	boolean	является ли заданная схема временной в другом сеансе?
pg_listening_channels()	setof text	имена каналов, по которым текущий сеанс принимает сигналы
pg_notification_queue_ usage()	double	занятая доля очереди асинхронных уведомлений (0-1)
pg_postmaster_start_time( )	timestamp with time zone	время запуска сервера
pg_safe_snapshot_ blocking_pids(int)	int[]	Идентификаторы процессов, не дающих серверному процессу с определённым ID получить безопасный снимок
pg_trigger_depth()	int	текущий уровень вложенности в триггерах Postgres Pro (0, если эта функция вызывается (прямо или косвенно) не из тела триггера)
session_user	name	имя пользователя сеанса
user	name	синоним current_user
version()	text	информация о версии PostgreSQL. Также можно прочитать версию в машинно-ориентированном виде, обратившись к переменной server_version_num.
pgpro_version()	text	информация о версии Postgres Pro
pgpro_edition()	text	название редакции Postgres Pro
pgpro_build()	text	идентификатор состояния исходного кода, из которого скомпилирован Postgres Pro

Примечание

Функции current_catalog, current_role, current_schema, current_user, session_user и user имеют особый синтаксический статус в SQL: они должны вызываться без скобок после имени. (Postgres Pro позволяет добавить скобки в вызове current_schema, но не других функций.)

Функция session_user обычно возвращает имя пользователя, установившего текущее соединение с базой данных, но суперпользователи могут изменить это имя, выполнив команду SET SESSION AUTHORIZATION. Функция current_user возвращает идентификатор пользователя, по которому будут проверяться его права. Обычно это тот же пользователь, что и пользователь сеанса, но его можно сменить с помощью SET ROLE. Этот идентификатор также меняется при выполнении функций с атрибутом SECURITY DEFINER. На языке Unix пользователь сеанса называется

«реальным», а текущий — «эффективным». Имена `current_role` и `user` являются синонимами `current_user`. (В стандарте SQL `current_role` и `current_user` имеют разное значение, но в Postgres Pro они не различаются, так как пользователи и роли объединены в единую сущность.)

Функция `current_schema` возвращает имя схемы, которая стоит первой в пути поиска (или NULL, если путь поиска пуст). Эта схема будет задействована при создании таблиц или других именованных объектов, если целевая схема не указана явно. Функция `current_schemas(boolean)` возвращает массив имён всех схем, находящихся в пути поиска. Её логический параметр определяет, будут ли включаться в результат неявно добавляемые в путь поиска системные схемы, такие как `pg_catalog`.

> **Примечание**
>
> Путь поиска можно изменить во время выполнения следующей командой:
>
> `SET search_path TO` *`схема`* `[,` *`схема`*`, ...]`

Функция `inet_client_addr` возвращает IP-адрес текущего клиента, `inet_client_port` — номер его порта, `inet_server_addr` — IP-адрес сервера, по которому он принял подключение клиента, а `inet_server_port` — соответствующий номер порта. Все эти функции возвращают NULL, если текущее соединение устанавливается через Unix-сокет.

Функция `pg_blocking_pids` возвращает массив идентификаторов процессов сеансов, которые блокирует серверный процесс с указанным идентификатором, либо пустой массив, если такой серверный процесс не найден или не заблокирован. Один серверный процесс блокирует другой, если он либо удерживает блокировку, конфликтующую с блокировкой, запрашиваемой серверным процессом (жёсткая блокировка), либо ждёт блокировки, которая вызвала бы конфликт с запросом блокировки заблокированного процесса и находится перед ней в очереди ожидания (мягкая блокировка). При распараллеливании запросов эта функция всегда выдаёт видимые клиентом идентификаторы процессов (то есть, результаты `pg_backend_pid`), даже если фактическая блокировка удерживается или ожидается дочерним рабочим процессом. Вследствие этого, в результатах могут оказаться дублирующиеся PID. Также заметьте, что когда конфликтующую блокировку удерживает подготовленная транзакция, в выводе этой функции она будет представлена нулевым ID процесса. Частые вызовы этой функции могут отразиться на производительности базы данных, так как ей нужен монопольный доступ к общему состоянию менеджера блокировок, хоть и на короткое время.

Функция `pg_conf_load_time` возвращает время (`timestamp with time zone`), когда в последний раз сервер загружал файлы конфигурации. (Если текущий сеанс начался раньше, она возвращает время, когда эти файлы были перезагружены для данного сеанса, так что в разных сеансах это значение может немного различаться. В противном случае это будет время, когда файлы конфигурации считал главный процесс.)

Функция `pg_current_logfile` возвращает в значении `text` путь к файлам журналов, в настоящее время используемым сборщиком сообщений. Этот путь состоит из каталога log_directory и имени файла журнала. Если сборщик сообщений отключён, возвращается значение `NULL`. Если ведутся несколько журналов в разных форматах, при вызове функции `pg_current_logfile` без аргументов возвращается путь файла, имеющего первый формат по порядку из следующего списка: stderr, csvlog. Если файл журнала имеет какой-то иной формат, возвращается `NULL`. Чтобы запросить файл в определённом формате, передайте либо csvlog, либо stderr в качестве значения необязательного параметра типа `text`. Если запрошенный формат не включён в log_destination, будет возвращено значение `NULL`. Функция `pg_current_logfile` отражает содержимое файла `current_logfiles`.

`pg_my_temp_schema` возвращает OID временной схемы текущего сеанса или 0, если такой нет (в рамках сеанса не создавались временные таблицы). `pg_is_other_temp_schema` возвращает true, если заданный OID относится к временной схеме другого сеанса. (Это может быть полезно,

например для исключения временных таблиц других сеансов из общего списка при просмотре таблиц базы данных.)

Функция `pg_listening_channels` возвращает набор имён каналов асинхронных уведомлений, на которые подписан текущий сеанс. Функция `pg_notification_queue_usage` возвращает долю от всего свободного пространства для уведомлений, в настоящее время занятую уведомлениями, ожидающими обработки, в виде значения `double` в диапазоне 0..1. За дополнительными сведениями обратитесь к LISTEN и NOTIFY.

`pg_postmaster_start_time` возвращает время (`timestamp with time zone`), когда был запущен сервер.

Функция `pg_safe_snapshot_blocking_pids` возвращает массив идентификаторов процессов сеансов, которые блокируют серверный процесс с указанным идентификатором (не дают получить ему безопасный снимок), либо пустой массив, если такой серверный процесс не найден или не заблокирован. Сеанс, выполняющий транзакцию уровня `SERIALIZABLE`, блокирует транзакцию `SERIALIZABLE READ ONLY DEFERRABLE`, не давая ей получить снимок, пока она не определит, что можно безопасно избежать установления предикатных блокировок. За дополнительными сведениями о сериализуемых и откладываемых транзакциях обратитесь к Подразделу 13.2.3. Частые вызовы этой функции могут отразиться на производительности базы данных, так как ей нужен доступ к общему состоянию менеджера предикатных блокировок, хоть и на короткое время.

Функция `version` возвращает строку, описывающую версию сервера Postgres Pro. Эту информацию также можно получить из переменной server_version или, в более машинно-ориентированном формате, из переменной server_version_num. При разработке программ следует использовать `server_version_num` (она появилась в версии 8.2) либо `PQserverVersion`, а не разбирать текстовую версию.

`pgpro_edition()` возвращает текстовую строку, описывающую редакцию Postgres Pro, например `standard` или `enterprise`.

`pgpro_version()` возвращает текстовую строку, описывающую версию сервера Postgres Pro.

В Таблице 9.61 перечислены функции, позволяющую пользователю программно проверить свои права доступа к объектам. Подробнее о правах можно узнать в Разделе 5.6.

Таблица 9.61. Функции для проверки прав доступа

Имя	Тип результата	Описание
`has_any_column_privilege ( user, table, privilege)`	`boolean`	имеет ли пользователь указанное право для какого-либо столбца таблицы
`has_any_column_privilege ( table, privilege)`	`boolean`	имеет ли текущий пользователь указанное право для какого-либо столбца таблицы
`has_column_privilege ( user, table, column, privilege)`	`boolean`	имеет ли пользователь указанное право для столбца
`has_column_privilege ( table, column, privilege)`	`boolean`	имеет ли текущий пользователь указанное право для столбца
`has_database_privilege ( user, database, privilege)`	`boolean`	имеет ли пользователь указанное право для базы данных
`has_database_privilege ( database, privilege)`	`boolean`	имеет ли текущий пользователь указанное право для базы данных

Имя	Тип результата	Описание
has_foreign_data_wrapper_privilege (*user*, *fdw*, *privilege*)	boolean	имеет ли пользователь указанное право для обёртки сторонних данных
has_foreign_data_wrapper_privilege (*fdw*, *privilege*)	boolean	имеет ли текущий пользователь указанное право для обёртки сторонних данных
has_function_privilege (*user*, *function*, *privilege*)	boolean	имеет ли пользователь указанное право для функции
has_function_privilege (*function*, *privilege*)	boolean	имеет ли текущий пользователь указанное право для функции
has_language_privilege (*user*, *language*, *privilege*)	boolean	имеет ли пользователь указанное право для языка
has_language_privilege (*language*, *privilege*)	boolean	имеет ли текущий пользователь указанное право для языка
has_schema_privilege (*user*, *schema*, *privilege*)	boolean	имеет ли пользователь указанное право для схемы
has_schema_privilege (*schema*, *privilege*)	boolean	имеет ли текущий пользователь указанное право для схемы
has_sequence_privilege (*user*, *sequence*, *privilege*)	boolean	имеет ли пользователь указанное право для последовательности
has_sequence_privilege (*sequence*, *privilege*)	boolean	имеет ли текущий пользователь указанное право для последовательности
has_server_privilege (*user*, *server*, *privilege*)	boolean	имеет ли пользователь указанное право для стороннего сервера
has_server_privilege (*server*, *privilege*)	boolean	имеет ли текущий пользователь указанное право для стороннего сервера
has_table_privilege (*user*, *table*, *privilege*)	boolean	имеет ли пользователь указанное право для таблицы
has_table_privilege (*table*, *privilege*)	boolean	имеет ли текущий пользователь указанное право для таблицы
has_tablespace_privilege (*user*, *tablespace*, *privilege*)	boolean	имеет ли пользователь указанное право для табличного пространства
has_tablespace_privilege (*tablespace*, *privilege*)	boolean	имеет ли текущий пользователь указанное право для табличного пространства
has_type_privilege (*user*, *type*, *privilege*)	boolean	имеет ли пользователь указанное право для типа
has_type_privilege (*type*, *privilege*)	boolean	имеет ли текущий пользователь указанное право для типа
pg_has_role (*user*, *role*, *privilege*)	boolean	имеет ли пользователь указанное право для роли
pg_has_role (*role*, *privilege*)	boolean	имеет ли текущий пользователь указанное право для роли

Имя	Тип результата	Описание
row_security_active (*table*)	boolean	включена ли для текущего пользователя защита на уровне строк для таблицы

has_table_privilege проверяет, может ли пользователь выполнять с таблицей заданные действия. В качестве идентификатора пользователя можно задать его имя, OID (pg_authid.oid) или public (это будет указывать на псевдороль PUBLIC). Если этот аргумент опущен, подразумевается текущий пользователь (current_user). Таблицу можно указать по имени или по OID. (Таким образом, фактически есть шесть вариантов функции has_table_privilege, различающихся по числу и типу аргументов.) Когда указывается имя объекта, его можно дополнить именем схемы, если это необходимо. Интересующее право доступа записывается в виде текста и может быть одним из следующих: SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES и TRIGGER. Дополнительно к названию права можно добавить WITH GRANT OPTION и проверить, разрешено ли пользователю передавать это право другим. Кроме того, в одном параметре можно перечислить несколько названий прав через запятую, и тогда функция возвратит true, если пользователь имеет одно из этих прав. (Регистр в названии прав не имеет значения, а между ними (но не внутри) разрешены пробельные символы.) Пара примеров:

```
SELECT has_table_privilege('myschema.mytable', 'select');
SELECT has_table_privilege('joe', 'mytable',
  'INSERT, SELECT WITH GRANT OPTION');
```

has_sequence_privilege проверяет, может ли пользователь выполнять заданные действия с последовательностью. В определении аргументов эта функция аналогична has_table_privilege. Допустимые для неё права складываются из USAGE, SELECT и UPDATE.

has_any_column_privilege проверяет, может ли пользователь выполнять заданные действия с каким-либо столбцом таблицы. В определении аргументов эта функция аналогична has_table_privilege, а допустимые права складываются из SELECT, INSERT, UPDATE и REFERENCES. Заметьте, что любое из этих прав, назначенное на уровне таблицы, автоматически распространяется на все её столбцы, так что has_any_column_privilege всегда возвращает true, если has_table_privilege даёт положительный ответ для тех же аргументов. Но has_any_column_privilege возвращает true ещё и тогда, когда право назначено только для некоторых столбцов.

has_column_privilege проверяет, может ли пользователь выполнять заданные действия со столбцом таблицы. В определении аргументов эта функция аналогична has_table_privilege, с небольшим дополнением: столбец можно задать по имени или номеру атрибута. Для неё допустимые права складываются из SELECT, INSERT, UPDATE и REFERENCES. Заметьте, что любое из этих прав, назначенное на уровне таблицы, автоматически распространяется на все столбцы таблицы.

has_database_privilege проверяет, может ли пользователь выполнять заданные действия с базой данных. В определении аргументов эта функция аналогична has_table_privilege. Для неё допустимые права складываются из CREATE, CONNECT и TEMPORARY (или TEMP, что равносильно TEMPORARY).

has_function_privilege проверяет, может ли пользователь обратиться к заданной функции. В определении аргументов эта функция аналогична has_table_privilege. Когда функция определяется не своим OID, а текстовой строкой, эта строка должна быть допустимой для вводимого значения типа regprocedure (см. Раздел 8.18). Для этой функции допустимо только право EXECUTE. Например:

```
SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');
```

has_foreign_data_wrapper_privilege проверяет, может ли пользователь обращаться к обёртке сторонних данных. В определении аргументов она аналогична has_table_privilege. Для неё допустимо только право USAGE.

has_language_privilege проверяет, может ли пользователь обращаться к процедурному языку. В определении аргументов эта функция аналогична has_table_privilege. Для неё допустимо только право USAGE.

has_schema_privilege проверяет, может ли пользователь выполнять заданные действия со схемой. В определении аргументов эта функция аналогична has_table_privilege. Для неё допустимые права складываются из CREATE и USAGE.

has_server_privilege проверяет, может ли пользователь обращаться к стороннему серверу. В определении аргументов она аналогична has_table_privilege. Для неё допустимо только право USAGE.

has_tablespace_privilege проверяет, может ли пользователь выполнять заданное действие в табличном пространстве. В определении аргументов эта функция аналогична has_table_privilege. Для неё допустимо только право CREATE.

has_type_privilege проверяет, может ли пользователь обратиться к типу определённым образом. Возможные аргументы аналогичны has_table_privilege. При указании типа текстовой строкой, а не по OID, допускаются те же входные значения, что и для типа данных regtype (см. Раздел 8.18). Для неё допустимо только право USAGE.

pg_has_role проверяет, может ли пользователь выполнять заданные действия с ролью. В определении аргументов эта функция аналогична has_table_privilege, за исключением того, что именем пользователя не может быть public. Для неё допустимые права складываются из MEMBER и USAGE. MEMBER обозначает прямое или косвенное членство в данной роли (то есть наличие права выполнить команду SET ROLE), тогда как USAGE показывает, что пользователь получает все права роли сразу, без SET ROLE.

row_security_active проверяет, включена ли защита на уровне строк для указанной таблицы в контексте и окружении текущего пользователя (current_user). Таблицу можно задать по имени или OID.

В Таблице 9.62 перечислены функции, определяющие *видимость* объекта с текущим путём поиска схем. К примеру, таблица считается видимой, если содержащая её схема включена в путь поиска и нет другой таблицы с тем же именем, которая была бы найдена по пути поиска раньше. Другими словами, к этой таблице можно будет обратиться просто по её имени, без явного указания схемы. Просмотреть список всех видимых таблиц можно так:

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

Таблица 9.62. Функции для определения видимости

Имя	Тип результата	Описание
pg_collation_is_visible(*collation_oid*)	boolean	видимо ли правило сортировки
pg_conversion_is_visible(*conversion_oid*)	boolean	видимо ли преобразование
pg_function_is_visible(*function_oid*)	boolean	видима ли функция
pg_opclass_is_visible(*opclass_oid*)	boolean	видим ли класс операторов
pg_operator_is_visible(*operator_oid*)	boolean	видим ли оператор
pg_opfamily_is_visible(*opclass_oid*)	boolean	видимо ли семейство операторов
pg_statistics_obj_is_visible(*stat_oid*)	boolean	видим ли объект статистики в пути поиска

Имя	Тип результата	Описание
pg_table_is_visible(*table_oid*)	boolean	видима ли таблица
pg_ts_config_is_visible(*config_oid*)	boolean	видима ли конфигурация текстового поиска
pg_ts_dict_is_visible(*dict_oid*)	boolean	видим ли словарь текстового поиска
pg_ts_parser_is_visible(*parser_oid*)	boolean	видим ли анализатор текстового поиска
pg_ts_template_is_visible(*template_oid*)	boolean	видим ли шаблон текстового поиска
pg_type_is_visible(*type_oid*)	boolean	видим ли тип (или домен)

Каждая из этих функций проверяет видимость объектов определённого типа. Заметьте, что pg_table_is_visible можно также использовать для представлений, материализованных представлений, индексов, последовательностей и сторонних таблиц; pg_type_is_visible можно также использовать и для доменов. Для функций и операторов объект считается видимым в пути поиска, если при просмотре пути не находится предшествующий ему другой объект с тем же именем *и типами аргументов*. Для классов операторов во внимание принимается и имя оператора, и связанный с ним метод доступа к индексу.

Всем этим функциям должен передаваться OID проверяемого объекта. Если вы хотите проверить объект по имени, удобнее использовать типы-псевдонимы OID (regclass, regtype, regprocedure, regoperator, regconfig или regdictionary), например:

```
SELECT pg_type_is_visible('myschema.widget'::regtype);
```

Заметьте, что проверять таким способом имена без указания схемы не имеет большого смысла — если имя удастся распознать, значит и объект будет видимым.

В Таблице 9.63 перечислены функции, извлекающие информацию из системных каталогов.

Таблица 9.63. Функции для обращения к системным каталогам

Имя	Тип результата	Описание
format_type(*type_oid* , *typemod*)	text	получает имя типа данных в формате SQL
pg_get_constraintdef(*constraint_oid*)	text	получает определение ограничения
pg_get_constraintdef(*constraint_oid* , *pretty_bool*)	text	получает определение ограничения
pg_get_expr(*pg_node_tree* , *relation_oid*)	text	декомпилирует внутреннюю форму выражения, в предположении, что все переменные в нём ссылаются на таблицу или отношение, указанное вторым параметром
pg_get_expr(*pg_node_tree* , *relation_oid* , *pretty_bool*)	text	декомпилирует внутреннюю форму выражения, в предположении, что все переменные в нём ссылаются на таблицу или отношение, указанное вторым параметром

Имя	Тип результата	Описание
pg_get_functiondef(*func_oid*)	text	получает определение функции
pg_get_function_arguments(*func_oid*)	text	получает список аргументов из определения функции (со значениями по умолчанию)
pg_get_function_identity_arguments(*func_oid*)	text	получает список аргументов, идентифицирующий функцию (без значений по умолчанию)
pg_get_function_result(*func_oid*)	text	получает предложение RETURNS для функции
pg_get_indexdef(*index_oid*)	text	получает команду CREATE INDEX для индекса
pg_get_indexdef(*index_oid* , *column_no* , *pretty_bool*)	text	получает команду CREATE INDEX для индекса или определение одного индексированного столбца, когда *column_no* не равен 0
pg_get_keywords()	setof record	получает список ключевых слов SQL по категориям
pg_get_ruledef(*rule_oid*)	text	получает команду CREATE RULE для правила
pg_get_ruledef(*rule_oid* , *pretty_bool*)	text	получает команду CREATE RULE для правила
pg_get_serial_sequence(*table_name* , *column_name*)	text	получает имя последовательности, используемой столбцом идентификации или столбцом serial
pg_get_statisticsobjdef(*statobj_oid*)	text	получает команду CREATE STATISTICS для объекта расширенной статистики
pg_get_triggerdef (*trigger_oid*)	text	получает команду CREATE [CONSTRAINT] TRIGGER для триггера
pg_get_triggerdef (*trigger_oid* ,*pretty_bool*)	text	получает команду CREATE [CONSTRAINT] TRIGGER для триггера
pg_get_userbyid(*role_oid*)	name	получает имя роли по заданному OID
pg_get_viewdef(*view_name*)	text	получает команду SELECT, определяющую представление или материализованное представление (*устаревшая* функция)
pg_get_viewdef(*view_name* , *pretty_bool*)	text	получает команду SELECT, определяющую представление или материализованное представление (*устаревшая* функция)

Имя	Тип результата	Описание
pg_get_viewdef(*view_oid*)	text	получает команду SELECT, определяющую представление или материализованное представление
pg_get_viewdef(*view_oid* , *pretty_bool*)	text	получает команду SELECT, определяющую представление или материализованное представление
pg_get_viewdef(*view_oid* , *wrap_column_int*)	text	получает команду SELECT, определяющую представление или материализованное представление; при необходимости разбивает строки с полями, выходящие за wrap_int символов, подразумевая форматированный вывод
pg_index_column_has_property(*index_oid* , *column_no* , *prop_name*)	boolean	проверяет, имеет ли столбец индекса заданное свойство
pg_index_has_property(*index_oid* , *prop_name*)	boolean	проверяет, имеет ли индекс заданное свойство
pg_indexam_has_property(*am_oid* , *prop_name*)	boolean	проверяет, имеет ли метод доступа индекса заданное свойство
pg_options_to_table(*reloptions*)	setof record	получает набор параметров хранилища в виде имя/значение
pg_tablespace_databases(*tablespace_oid*)	setof oid	получает или устанавливает OID баз данных, объекты которых содержатся в заданном табличном пространстве
pg_tablespace_location(*tablespace_oid*)	text	получает путь в файловой системе к местоположению заданного табличного пространства
pg_typeof(*any*)	regtype	получает тип данных любого значения
collation for (*any*)	text	получает правило сортировки для аргумента
to_regclass(*rel_name*)	regclass	получает OID указанного отношения
to_regproc(*func_name*)	regproc	получает OID указанной функции
to_regprocedure(*func_name*)	regprocedure	получает OID указанной функции
to_regoper(*имя_оператора*)	regoper	получает OID указанного оператора
to_regoperator(*имя_оператора*)	regoperator	получает OID указанного оператора

Имя	Тип результата	Описание
to_regtype(*type_name*)	regtype	получает OID указанного типа
to_regnamespace(*schema_name*)	regnamespace	получает OID указанной схемы
to_regrole(*role_name*)	regrole	получает OID указанной роли

format_type возвращает в формате SQL имя типа данных, определяемого по OID и, возможно, модификатору типа. Если модификатор неизвестен, вместо него можно передать NULL.

pg_get_keywords возвращает таблицу с ключевыми словами SQL, которые воспринимает сервер. Столбец word содержит ключевое слово, а catcode — код категории: U — не зарезервировано, C — имя столбца, T — имя типа или функции, R — зарезервировано. Столбец catdesc содержит возможно локализованное описание категории.

pg_get_constraintdef, pg_get_indexdef, pg_get_ruledef, pg_get_statisticsobjdef и pg_get_triggerdef восстанавливают команду, создававшую заданное ограничение, индекс, правило, объект статистики или триггер, соответственно. (Учтите, что они возвращают не изначальный текст команды, а результат декомпиляции.) pg_get_expr декомпилирует внутреннюю форму отдельного выражения, например значения по умолчанию для столбца. Это может быть полезно для изучения содержимого системных каталогов. Если выражение может содержать переменные, укажите во втором параметре OID отношения, на который они ссылаются; если таких переменных нет, вместо OID можно передать 0. pg_get_viewdef восстанавливает запрос SELECT, определяющий представление. Многие из этих функций имеют две версии, одна из которых позволяет получить форматированный вывод (с параметром pretty_bool). Форматированный текст легче читается, но нет гарантии, что он будет всегда восприниматься одинаково будущими версиями Postgres Pro; поэтому не следует применять форматирование при выгрузке метаданных. Если в параметре pretty_bool передаётся false, эта версия функции выдаёт тот же результат, что и версия без параметров.

pg_get_functiondef возвращает полный оператор CREATE OR REPLACE FUNCTION для заданной функции. pg_get_function_arguments возвращает список аргументов функции, в виде достаточном для включения в команду CREATE FUNCTION. pg_get_function_result в дополнение возвращает готовое предложение RETURNS для функции. pg_get_function_identity_arguments возвращает список аргументов, достаточный для однозначной идентификации функции, в форме, допустимой, например для команды ALTER FUNCTION. Значения по умолчанию в этой форме опускаются.

pg_get_serial_sequence возвращает имя последовательности, связанной со столбцом, либо NULL, если такой последовательности нет. Для столбца идентификации это будет последовательность, связанная с ним внутренним образом. Для столбцов, имеющих один из последовательных типов (serial, smallserial, bigserial), это последовательность, созданная объявлением данного столбца. В последнем случае эту связь можно изменить или разорвать, воспользовавшись командой ALTER SEQUENCE OWNED BY. (Возможно, эту функцию стоило назвать pg_get_owned_sequence; её существующее имя отражает тот факт, что она обычно используется со столбцами serial или bigserial.) В первом параметре функции указывается имя таблицы (возможно, дополненное схемой), а во втором имя столбца. Так как первый параметр может содержать имя схемы и таблицы, он воспринимается не как идентификатор в кавычках и поэтому по умолчанию приводится к нижнему регистру, тогда как имя столбца во втором параметре воспринимается как заключённое в кавычки и в нём регистр символов сохраняется. Эта функция возвращает имя в виде, пригодном для передачи функциям, работающим с последовательностями (см. Раздел 9.16). Обычно она применяется для получения текущего значения последовательности для столбца идентификации или последовательного столбца, например:

```
SELECT currval(pg_get_serial_sequence('sometable', 'id'));
```

pg_get_userbyid получает имя роли по её OID.

Функции pg_index_column_has_property, pg_index_has_property и pg_indexam_has_property показывают, обладает ли указанный столбец индекса, индекс или метод доступа индекса заданным

свойством. Они возвращают NULL, если имя свойства неизвестно или неприменимо к конкретному объекту, либо если OID или номер столбца не указывают на действительный объект. Описание свойств столбцов вы можете найти в Таблице 9.64, свойства индексов описаны в Таблице 9.65, а свойства методов доступа — в Таблица 9.66. (Заметьте, что методы доступа, реализуемые расширениями, могут определять для своих индексов дополнительные имена свойств.)

Таблица 9.64. Свойства столбца индекса

Имя	Описание
asc	Сортируется ли столбец по возрастанию при сканировании вперёд?
desc	Сортируется ли столбец по убыванию при сканировании вперёд?
nulls_first	Выдаются ли NULL в начале при сканирования вперёд?
nulls_last	Выдаются ли NULL в конце при сканировании вперёд?
orderable	Связан ли со столбцом некоторый порядок сортировки?
distance_orderable	Может ли столбец сканироваться по порядку оператором «расстояния», например, ORDER BY столбец <-> константа ?
returnable	Может ли значение столбца быть получено при сканировании только индекса?
search_array	Поддерживает ли столбец внутренними средствами поиск столбец = ANY(массив) ?
search_nulls	Поддерживает ли столбец поиск IS NULL и IS NOT NULL?

Таблица 9.65. Свойства индекса

Имя	Описание
clusterable	Может ли индекс использоваться в команде CLUSTER?
index_scan	Поддерживает ли индекс простое сканирование (не по битовой карте)?
bitmap_scan	Поддерживает ли индекс сканирование по битовой карте?
backward_scan	Может ли в процессе сканирования меняться направление (для поддержки перемещения курсора FETCH BACKWARD без необходимости материализации)?

Таблица 9.66. Свойства метода доступа индекса

Имя	Описание
can_order	Поддерживает ли метод доступа ASC, DESC и связанные ключевые слова в CREATE INDEX?
can_unique	Поддерживает ли метод доступа уникальные индексы?
can_multi_col	Поддерживает ли метод доступа индексы по нескольким столбцам?

Имя	Описание
can_exclude	Поддерживает ли метод доступа ограничения-исключения?

pg_options_to_table возвращает набор параметров хранилища в виде пар (*имя_параметра*/*значение_параметра*), когда ей передаётся pg_class.reloptions или pg_attribute.attoptions.

pg_tablespace_databases позволяет изучить содержимое табличного пространства. Она возвращает набор OID баз данных, объекты которых размещены в этом табличном пространстве. Если эта функция возвращает строки, это означает, что табличное пространство не пустое и удалить его нельзя. Какие именно объекты находятся в табличном пространстве, можно узнать, подключаясь к базам данных, OID которых сообщила pg_tablespace_databases, и анализируя их каталоги pg_class.

pg_typeof возвращает OID типа данных для переданного значения. Это может быть полезно для разрешения проблем или динамического создания SQL-запросов. Эта функция объявлена как возвращающая тип regtype, который является псевдонимом типа OID (см. Раздел 8.18); это означает, что значение этого типа можно сравнивать как OID, но выводится оно как название типа. Например:

```
SELECT pg_typeof(33);

 pg_typeof
-----------
 integer
(1 row)

SELECT typlen FROM pg_type WHERE oid = pg_typeof(33);
 typlen
--------
      4
(1 row)
```

Выражение collation for возвращает правило сортировки для переданного значения. Например:

```
SELECT collation for (description) FROM pg_description LIMIT 1;
 pg_collation_for
------------------
 "default"
(1 row)

SELECT collation for ('foo' COLLATE "de_DE");
 pg_collation_for
------------------
 "de_DE"
(1 row)
```

Это значение может быть заключено в кавычки и дополнено схемой. Если для выражения аргумента нет правила сортировки, возвращается значение NULL. Если же правила сортировки не применимы для типа аргумента, происходит ошибка.

Функции to_regclass, to_regproc, to_regprocedure, to_regoper, to_regoperator, to_regtype, to_regnamespace и to_regrole преобразуют имена отношений, функций, операторов, типов, схем и ролей (заданных значением text) в объекты типа regclass, regproc, regprocedure, regoper, regoperator, regtype, regnamespace и regrole, соответственно. Перечисленные функции отличаются от явных приведений к этим типам тем, что они не принимают числовые OID и возвращают NULL вместо ошибки, если имя не найдено (или, в случае с to_regproc и to_regoper, если данному имени соответствуют несколько объектов).

В Таблице 9.67 перечислены функции, связанные с идентификацией и адресацией объектов баз данных.

Таблица 9.67. Функции получения информации и адресации объектов

Имя	Тип результата	Описание
pg_describe_object(*classid* oid, *objid* oid, *objsubid* integer)	text	получает описание объекта базы данных
pg_identify_object(*classid* oid, *objid* oid, *objsubid* integer)	*type* text, *schema* text, *name* text, *identity* text	получает идентификатор объекта базы данных
pg_identify_object_as_address(*classid* oid, *objid* oid, *objsubid* integer)	*type* text, *object_names* text[], *object_args* text[]	получает внешнее представление адреса объекта базы данных
pg_get_object_address(*type* text, *name* text[], *аргументы* text[])	*classid* oid, *objid* oid, *objsubid* integer	получает адрес объекта базы данных из его внешнего представления

pg_describe_object возвращает текстовое описание объекта БД, идентифицируемого по OID каталога, OID объекта и ID подобъекта (например, номер столбца в таблице; ID подобъекта равен нулю для объекта в целом). Это описание предназначено для человека и может переводиться, в зависимости от конфигурации сервера. С помощью этой функции, например, можно узнать, что за объект хранится в каталоге pg_depend.

pg_identify_object возвращает запись, содержащую достаточно информации для однозначной идентификации объекта БД по OID каталога, OID объекта и ID подобъекта. Эта информация предназначена для машины и поэтому никогда не переводится. Столбец *type* содержит тип объекта БД; *schema* — имя схемы, к которой относится объект (либо NULL для объектов, не относящихся к схемам); *name* — имя объекта, при необходимости в кавычках, которое присутствует, только если оно (возможно, вместе со схемой) однозначно идентифицирует объект (в противном случае NULL); *identity* — полный идентификатор объекта, точный формат которого зависит от типа объекта, а каждая его часть дополняется схемой и заключается в кавычки, если требуется.

pg_identify_object_as_address возвращает запись, содержащую достаточно информации для однозначной идентификации объекта БД по OID каталога, OID объекта и ID подобъекта. Выдаваемая информация не зависит от текущего сервера, то есть по ней можно идентифицировать одноимённый объект на другом сервере. Поле *type* содержит тип объекта БД, а *object_names* и *object_args* — текстовые массивы, в совокупности формирующие ссылку на объект. Эти три значения можно передать функции pg_get_object_address, чтобы получить внутренний адрес объекта. Данная функция является обратной к pg_get_object_address.

pg_get_object_address возвращает запись, содержащую достаточно информации для уникальной идентификации объекта БД по его типу и массивам имён и аргументов. В ней возвращаются значения, которые используются в системных каталогах, например pg_depend, и могут передаваться в другие системные функции, например pg_identify_object или pg_describe_object. Поле *classid* содержит OID системного каталога, к которому относится объект; *objid* — OID самого объекта, а *objsubid* — идентификатор подобъекта или 0 в случае его отсутствия. Эта функция является обратной к pg_identify_object_as_address.

Функции, перечисленные в Таблице 9.68, извлекают комментарии, заданные для объектов с помощью команды COMMENT. Если найти комментарий для заданных параметров не удаётся, они возвращают NULL.

Таблица 9.68. Функции получения комментариев

Имя	Тип результата	Описание
col_description(*table_oid* , *column_number*)	text	получает комментарий для столбца таблицы
obj_description(*object_oid* , *catalog_name*)	text	получает комментарий для объекта базы данных
obj_description(*object_oid*)	text	получает комментарий для объекта базы данных (*устаревшая* форма)
shobj_description(*object_oid* , *catalog_name*)	text	получает комментарий для разделяемого объекта баз данных

col_description возвращает комментарий для столбца с заданным номером в таблице с указанным OID. (obj_description нельзя использовать для столбцов таблицы, так столбцы не имеют собственных OID.)

Функция obj_description с двумя параметрами возвращает комментарий для объекта, имеющего заданный OID и находящегося в указанном системном каталоге. Например, obj_description(123456,'pg_class') вернёт комментарий для таблицы с OID 123456. Форма obj_description с одним параметром принимает только OID. Она является устаревшей, так как значения OID могут повторяться в разных системных каталогах, и поэтому она может возвращать комментарий для другого объекта.

shobj_description работает подобно obj_description, но она получает комментарии для разделяемых объектов. Некоторые системные каталоги являются глобальными для всех баз данных в кластере и описания объектов в них также хранятся глобально.

Функции, перечисленные в Таблице 9.69, выдают информацию о транзакциях сервера в форме во внешнем представлении. В основном эти функции используются, чтобы определить, какие транзакции были зафиксированы между двумя снимками состояния.

Таблица 9.69. Идентификаторы транзакций и снимков состояния

Имя	Тип результата	Описание
txid_current()	bigint	получает идентификатор текущей транзакции и присваивает новый, если текущая транзакция его не имеет
txid_current_if_assigned( )	bigint	работает аналогично txid_current() , но возвращает NULL, не присваивая новый идентификатор транзакции, если он ещё не присвоен
txid_current_snapshot()	txid_snapshot	получает код текущего снимка
txid_snapshot_xip(*txid_snapshot*)	setof bigint	возвращает идентификаторы выполняющихся транзакций в снимке
txid_snapshot_xmax(*txid_snapshot*)	bigint	возвращает значение xmax для заданного снимка
txid_snapshot_xmin(*txid_snapshot*)	bigint	возвращает значение xmin для заданного снимка

Имя	Тип результата	Описание
txid_visible_in_snapshot(*bigint*, *txid_snapshot*)	boolean	видима ли транзакция с указанным идентификатором в данном снимке? (коды подтранзакций не поддерживаются)
txid_status(*bigint*)	text	возвращает состояние заданной транзакции — committed (зафиксирована), aborted (прервана), in progress (выполняется) или NULL, если идентификатор транзакции слишком старый

Внутренний тип идентификаторов транзакций (xid) имеет размер 32 бита, поэтому они повторяются через 4 миллиарда транзакций. Однако эти функции выдают 64-битные значения, дополненные счётчиком «эпохи», так что эти значения останутся уникальными на протяжении всей жизни сервера. Используемый этими функциями тип данных txid_snapshot сохраняет информацию о видимости транзакций в определённый момент времени. Его состав описан в Таблице 9.70.

Таблица 9.70. Состав информации о снимке

Имя	Описание
xmin	Идентификатор самой ранней транзакции (txid) из активных. Все предыдущие транзакции либо зафиксированы и видимы, либо отменены и мертвы.
xmax	Первый txid из ещё не назначенных. На момент снимка не было запущенных (а значит и видимых) транзакций с идентификатором, большим или равным данному.
xip_list	Список идентификаторов транзакций, активных в момент снимка. Он включает только идентификаторы с номерами от xmin до xmax; хотя уже могут быть транзакции с идентификаторами больше xmax. Если в этом списке не оказывается идентификатора транзакции xmin <= txid < xmax, это означает, что она уже не выполнялась к моменту снимка и, таким образом, видима или мертва, в зависимости от типа завершения. Идентификаторы подтранзакций в этот список не включаются.

В текстовом виде txid_snapshot представляется как *xmin*:*xmax*:*xip_list*. Например, 10:20:10,14,15 означает xmin=10, xmax=20, xip_list=10, 14, 15.

Функция txid_status(bigint) выдаёт состояние фиксации последней транзакции. Проверяя его, приложения могут определить, была ли транзакция зафиксирована или прервана, когда приложение отключается от сервера баз данных в процессе выполнения COMMIT. Состояние транзакции может быть следующим: in progress (выполняется), committed (зафиксирована) и aborted (прервана), если только транзакция не настолько стара, чтобы система удалила информация о её состоянии фиксирования. Если же она так стара, что упоминаний о ней в системе не осталось и информация о фиксации потеряна, эта функция возвращает NULL. Заметьте, что для

подготовленных транзакций возвращается состояние `in progress`; приложения должны проверять `pg_prepared_xacts`, если им нужно определить, является ли транзакция «txid» подготовленной.

Функции, показанные в Таблице 9.71, выдают информацию об уже зафиксированных транзакциях. Они возвращают полезные данные, только когда включён параметр конфигурации track_commit_timestamp и только для транзакций, зафиксированных после его включения.

Таблица 9.71. Информация о фиксации транзакций

Имя	Тип результата	Описание
`pg_xact_commit_timestamp(` *xid*`)`	`timestamp with time zone`	выдаёт время фиксации транзакции
`pg_last_committed_xact()`	*xid* `xid`, *timestamp* `timestamp with time zone`	выдаёт идентификатор и время фиксации транзакции, зафиксированной последней

Функции, перечисленные в Таблице 9.72, выводят свойства, установленные командой `initdb`, например, версию каталога. Они также выводят сведения о работе журнала предзаписи и контрольных точках. Эта информация относится ко всему кластеру, а не к отдельной базе данных. Данные функции выдают практически всю ту же информацию, и из того же источника, что и pg_controldata, но в форме, более подходящей для функций SQL.

Таблица 9.72. Функции управления данными

Имя	Тип результата	Описание
`pg_control_checkpoint()`	`record`	Возвращает информацию о состоянии текущей контрольной точки.
`pg_control_system()`	`record`	Возвращает информацию о текущем состоянии управляющего файла.
`pg_control_init()`	`record`	Возвращает информацию о состоянии инициализации кластера.
`pg_control_recovery()`	`record`	Возвращает информацию о состоянии восстановления.

Функция `pg_control_checkpoint` возвращает запись, показанную в Таблице 9.73.

Таблица 9.73. Столбцы результата `pg_control_checkpoint`

Имя столбца	Тип данных
`checkpoint_lsn`	`pg_lsn`
`prior_lsn`	`pg_lsn`
`redo_lsn`	`pg_lsn`
`redo_wal_file`	`text`
`timeline_id`	`integer`
`prev_timeline_id`	`integer`
`full_page_writes`	`boolean`
`next_xid`	`text`
`next_oid`	`oid`
`next_multixact_id`	`xid`

Имя столбца	Тип данных
next_multi_offset	xid
oldest_xid	xid
oldest_xid_dbid	oid
oldest_active_xid	xid
oldest_multi_xid	xid
oldest_multi_dbid	oid
oldest_commit_ts_xid	xid
newest_commit_ts_xid	xid
checkpoint_time	timestamp with time zone

Функция pg_control_system возвращает запись, показанную в Таблице 9.74.

Таблица 9.74. Столбцы результата pg_control_system

Имя столбца	Тип данных
pg_control_version	integer
catalog_version_no	integer
system_identifier	bigint
pg_control_last_modified	timestamp with time zone
pg_control_edition	text

Функция pg_control_init возвращает запись, показанную в Таблице 9.75.

Таблица 9.75. Столбцы результата pg_control_init

Имя столбца	Тип данных
max_data_alignment	integer
database_block_size	integer
blocks_per_segment	integer
wal_block_size	integer
bytes_per_wal_segment	integer
max_identifier_length	integer
max_index_columns	integer
max_toast_chunk_size	integer
large_object_chunk_size	integer
float4_pass_by_value	boolean
float8_pass_by_value	boolean
data_page_checksum_version	integer
icu_version	text

> **Примечание**
>
> icu_version возвращает NULL, если Postgres Pro собран без поддержки ICU.

Функция pg_control_recovery возвращает запись, показанную в Таблице 9.76

Таблица 9.76. Столбцы результата pg_control_recovery

Имя столбца	Тип данных
min_recovery_end_lsn	pg_lsn
min_recovery_end_timeline	integer
backup_start_lsn	pg_lsn
backup_end_lsn	pg_lsn
end_of_backup_record_required	boolean

9.26. Функции для системного администрирования

Функции, описанные в этом разделе, предназначены для контроля и управления сервером Postgres Pro.

9.26.1. Функции для управления конфигурацией

В Таблице 9.77 показаны функции, позволяющие получить и изменить значения параметров конфигурации выполнения.

Таблица 9.77. Функции для управления конфигурацией

Имя	Тип результата	Описание
current_setting(*setting_name* [, *missing_ok*])	text	получает текущее значение параметра
set_config(*setting_name* , *new_value* , *is_local*)	text	устанавливает новое значение параметра и возвращает его

Функция current_setting выдаёт текущее значение параметра *setting_name*. Она соответствует стандартной SQL-команде SHOW. Пример использования:

```
SELECT current_setting('datestyle');

 current_setting
-----------------
 ISO, MDY
(1 row)
```

Если параметра с именем *setting_name* нет, функция current_setting выдаёт ошибку, если только дополнительно не передан параметр *missing_ok* со значением true.

set_config устанавливает для параметра *setting_name* значение *new_value*. Если параметр *is_local* равен true, новое значение будет действовать только в рамках текущей транзакции. Чтобы это значение действовало на протяжении текущего сеанса, ему нужно присвоить false. Эта функция соответствует SQL-команде SET. Пример использования:

```
SELECT set_config('log_statement_stats', 'off', false);

 set_config
------------
 off
(1 row)
```

9.26.2. Функции для передачи сигналов серверу

Функции, перечисленные в Таблице 9.78, позволяют передавать управляющие сигналы другим серверным процессам. Вызывать эти функции по умолчанию разрешено только суперпользователям, но доступ к ним можно дать и другим пользователям командой GRANT, кроме явно отмеченных исключений.

Таблица 9.78. Функции для передачи сигналов серверу

Имя	Тип результата	Описание
pg_cancel_backend(*pid* int)	boolean	Отменяет текущий запрос в обслуживающем процессе. Это действие разрешается и ролям, являющимся членами роли, обслуживающий процесс которой затрагивается, и ролям, которым дано право pg_signal_backend ; однако только суперпользователям разрешено воздействовать на обслуживающие процессы других суперпользователей.
pg_reload_conf()	boolean	Даёт команду серверным процессам перегрузить конфигурацию
pg_rotate_logfile()	boolean	Прокручивает журнал сообщений сервера
pg_terminate_backend(*pid* int)	boolean	Завершает обслуживающий процесс. Это действие разрешается и ролям, являющимся членами роли, обслуживающий процесс которой прерывается, и ролям, которым дано право pg_signal_backend ; однако только суперпользователям разрешено прерывать обслуживающие процессы других суперпользователей.

Каждая из этих функций возвращает true при успешном завершении и false в противном случае.

pg_cancel_backend и pg_terminate_backend передают сигналы (SIGINT и SIGTERM, соответственно) серверному процессу с заданным кодом PID. Код активного процесса можно получить из столбца pid представления pg_stat_activity или просмотрев на сервере процессы с именем postgres (используя ps в Unix или Диспетчер задач в Windows). Роль пользователя активного процесса можно узнать в столбце usename представления pg_stat_activity.

pg_reload_conf отправляет сигнал SIGHUP главному серверному процессу, который командует всем подчинённым процессам перезагрузить файлы конфигурации.

pg_rotate_logfile указывает менеджеру журнала сообщений немедленно переключиться на новый файл. Это имеет смысл, только когда работает встроенный сборщик сообщений, так как без него подпроцесс менеджера журнала не запускается.

9.26.3. Функции управления резервным копированием

Функции, перечисленные в Таблице 9.79, предназначены для выполнения резервного копирования «на ходу». Эти функции нельзя выполнять во время восстановления (за исключением

немонопольных вариантов pg_start_backup и pg_stop_backup, а также функций pg_is_in_backup, pg_backup_start_time и pg_wal_lsn_diff).

Таблица 9.79. Функции управления резервным копированием

Имя	Тип результата	Описание
pg_create_restore_point(*name* text)	pg_lsn	Создаёт именованную точку для восстановления (по умолчанию разрешено только суперпользователям, но право на её выполнение (EXECUTE) можно дать и другим пользователям)
pg_current_wal_flush_lsn()	pg_lsn	Получает текущую позицию сброса данных в журнале предзаписи
pg_current_wal_insert_lsn()	pg_lsn	Получает текущую позицию добавления в журнале предзаписи
pg_current_wal_lsn()	pg_lsn	Получает текущую позицию записи в журнале предзаписи
pg_start_backup(*label* text [, *fast* boolean [, *exclusive* boolean]])	pg_lsn	Подготавливает сервер к резервному копированию (по умолчанию разрешено только суперпользователям, но право на её выполнение (EXECUTE) можно дать и другим пользователям)
pg_stop_backup()	pg_lsn	Завершает монопольное резервное копирование (по умолчанию разрешено только суперпользователям, но право на её выполнение (EXECUTE) можно дать и другим пользователям)
pg_stop_backup(*exclusive* boolean [, *wait_for_archive* boolean])	setof record	Завершает монопольное или немонопольное резервное копирование (по умолчанию разрешено только суперпользователям, но право на её выполнение (EXECUTE) можно дать и другим пользователям)
pg_is_in_backup()	bool	Возвращает true в процессе исключительного резервного копирования
pg_backup_start_time()	timestamp with time zone	Получает время запуска выполняющегося исключительного резервного копирования
pg_switch_wal()	pg_lsn	Инициирует переключение на новый файл журнала предзаписи (по умолчанию разрешено только суперпользователям, но право

Имя	Тип результата	Описание
		на её выполнение (EXECUTE) можно дать и другим пользователям)
pg_walfile_name(*lsn* pg_lsn)	text	Получает из позиции в журнале предзаписи имя соответствующего файла
pg_walfile_name_offset(*lsn* pg_lsn)	text, integer	Получает из позиции в журнале предзаписи имя соответствующего файла и десятичное смещение в нём
pg_wal_lsn_diff(*lsn* pg_lsn , *lsn* pg_lsn)	numeric	Вычисляет разницу между двумя позициями в журнале предзаписи

pg_start_backup принимает произвольную заданную пользователем метку резервной копии. (Обычно это имя файла, в котором будет сохранена резервная копия.) При копировании в монопольном режиме эта функция записывает файл метки (backup_label) и, если есть ссылки в каталоге pg_tblspc/, файл карты табличных пространств (tablespace_map) в каталог данных кластера БД, выполняет процедуру контрольной точки, а затем возвращает в текстовом виде начальную позицию в журнале предзаписи для данной резервной копии. Результат этой функции может быть полезен, но если он не нужен, его можно просто игнорировать. При копировании в немонопольном режиме содержимое этих файлов выдаётся функцией pg_stop_backup и должно быть записано в архивную копию вызывающим субъектом.

```
postgres=# select pg_start_backup('label_goes_here');
 pg_start_backup
-----------------
 0/D4445B8
(1 row)
```

У этой функции есть также второй, необязательный параметр типа boolean. Если он равен true, pg_start_backup начнёт работу максимально быстро. При этом будет немедленно выполнена процедура контрольной точки, что может повлечь массу операций ввода/вывода и затормозить параллельные запросы.

При монопольном копировании функция pg_stop_backup удаляет файл метки (и, если существует, файл tablespace_map), созданный функцией pg_start_backup. При немонопольном копировании содержимое backup_label и tablespace_map возвращается в результате функции и должно быть записано в файлы в архиве (а не в каталоге данных). Также есть необязательный второй параметр типа boolean. Если он равен false, pg_stop_backup завершится сразу после окончания резервного копирования, не дожидаясь архивации WAL. Это поведение полезно только для программ резервного копирования, которые независимо осуществляют архивацию WAL. Если же WAL не заархивируется, резервная копия может оказаться неполной, а значит, бесполезной. Когда он равен true, pg_stop_backup будет ждать выполнения архивации WAL, если архивирование включено; для резервного сервера это означает, что ожидание возможно только при условии archive_mode = always. Если активность записи на ведущем сервере невысока, может иметь смысл выполнить на нём pg_switch_wal для немедленного переключения сегмента.

При выполнении на ведущем эта функция также создаёт файл истории резервного копирования в области архива журнала предзаписи. В этом файле для данной резервной копии сохраняется метка, заданная при вызове pg_start_backup, начальная и конечная позиция в журнале предзаписи, а также время начала и окончания. Возвращает данная функция позицию окончания резервной копии в журнале предзаписи (которую тоже можно игнорировать). После записи конечной позиции текущая позиция автоматически перемещается к следующему файлу журнала предзаписи, чтобы файл конечной позиции можно было немедленно архивировать для завершения резервного копирования.

`pg_switch_wal` производит переключение на следующий файл журнала предзаписи, что позволяет архивировать текущий (в предположении, что архивация выполняется непрерывно). Эта функция возвращает конечную позицию в журнале предзаписи (в только что законченном файле журнала) + 1. Если с момента последнего переключения файлов не было активности, отражающейся в журнале предзаписи, `pg_switch_wal` не делает ничего и возвращает начальную позицию в файле журнала предзаписи, используемом в данный момент.

`pg_create_restore_point` создаёт именованную запись в журнале предзаписи, которую можно использовать как цель при восстановлении, и возвращает соответствующую позицию в журнале предзаписи. Затем полученное имя можно присвоить параметру recovery_target_name, указав тем самым точку, до которой будет выполняться восстановление. Учтите, что если вы создадите несколько точек восстановления с одним именем, восстановление будет остановлено на первой точке с этим именем.

`pg_current_wal_lsn` выводит текущую позицию записи в журнале предзаписи в том же формате, что и вышеописанные функции. `pg_current_wal_insert_lsn` подобным образом выводит текущую позицию добавления, а `pg_current_wal_flush_lsn` — позицию сброса данных журнала. Позицией добавления называется «логический» конец журнала предзаписи в любой момент времени, тогда как позиция записи указывает на конец данных, фактически вынесённых из внутренних буферов сервера, а позиция сброса показывает, до какого места данные гарантированно сохранены в надёжном хранилище. Позиция записи отмечает конец данных, которые может видеть снаружи внешний процесс, и именно она представляет интерес при копировании частично заполненных файлов журнала. Позиция добавления и позиция сброса выводятся в основном для отладки серверной части. Все эти операции выполняются в режиме «только чтение» и не требуют прав суперпользователя.

Из результатов всех описанных выше функций можно получить имя файла журнала предзаписи и смещение в нём, используя функцию `pg_walfile_name_offset`. Например:

```
postgres=# SELECT * FROM pg_walfile_name_offset(pg_stop_backup());
        file_name         | file_offset
--------------------------+-------------
 00000001000000000000000D |     4039624
(1 row)
```

Подобным образом, `pg_walfile_name` извлекает только имя файла журнала предзаписи. Когда заданная позиция в журнале предзаписи находится ровно на границе файлов, обе эти функции возвращают имя предыдущего файла. Обычно это поведение предпочтительно при архивировании журнала предзаписи, так как именно предыдущий файл является последним подлежащим архивации.

`pg_wal_lsn_diff` вычисляет разницу в байтах между двумя позициями в журнале предзаписи. Полученный результат можно использовать с `pg_stat_replication` или другими функциями, перечисленными в Таблице 9.79, для определения задержки репликации.

Подробнее практическое применение этих функций описывается в Разделе 24.3.

9.26.4. Функции управления восстановлением

Функции, приведённые в Таблице 9.80, предоставляют сведения о текущем состоянии ведомого сервера. Эти функции могут выполняться, как во время восстановления, так и в обычном режиме работы.

Таблица 9.80. Функции для получения информации о восстановлении

Имя	Тип результата	Описание
`pg_is_in_recovery()`	`bool`	Возвращает true в процессе восстановления.
`pg_last_wal_receive_lsn()`	`pg_lsn`	Получает позицию последней записи журнала предзаписи,

Имя	Тип результата	Описание
		полученной и записанной на диск в процессе потоковой репликации. Пока выполняется потоковая репликация, эта позиция постоянно увеличивается. По окончании восстановления она останавливается на записи WAL, полученной и записанной на диск последней. Если потоковая репликация отключена или ещё не запускалась, функция возвращает NULL.
`pg_last_wal_replay_lsn()`	`pg_lsn`	Получает позицию последней записи журнала предзаписи, воспроизведённой при восстановлении. В процессе восстановления эта позиция постоянно увеличивается. По окончании восстановления она останавливается на записи WAL, которая была восстановлена последней. Если сервер был запущен не в режиме восстановления, эта функция возвращает NULL.
`pg_last_xact_replay_timestamp()`	`timestamp with time zone`	Получает отметку времени последней транзакции, воспроизведённой при восстановлении. Это время, когда на главном сервере произошла фиксация или откат записи WAL для этой транзакции. Если в процессе восстановления не была воспроизведена ни одна транзакция, эта функция возвращает NULL. В противном случае это значение постоянно увеличивается в процессе восстановления. По окончании восстановления оно останавливается на транзакции, которая была восстановлена последней. Если сервер был запущен не в режиме восстановления, эта функция возвращает NULL.

Функции, перечисленные в Таблице 9.81 управляют процессом восстановления. Вызывать их в другое время нельзя.

Таблица 9.81. Функции управления восстановлением

Имя	Тип результата	Описание
`pg_is_wal_replay_paused()`	`bool`	Возвращает true, если восстановление приостановлено.
`pg_wal_replay_pause()`	`void`	Немедленно приостанавливает восстановление (по умолчанию разрешено только суперпользователям, но право на её выполнение (EXECUTE) можно дать и другим пользователям).
`pg_wal_replay_resume()`	`void`	Перезапускает восстановление, если оно было приостановлено (по умолчанию разрешено только суперпользователям, но право на её выполнение (EXECUTE) можно дать и другим пользователям).

Когда восстановление приостановлено, запись изменений в базу не производится. Если она находится в «горячем резерве», все последующие запросы будут видеть один согласованный снимок базы данных и до продолжения восстановления конфликты запросов исключаются.

Когда потоковая репликация выключена, пауза при восстановлении может длиться сколь угодно долго без каких-либо проблем. Если же запущена потоковая репликация, новые записи WAL продолжат поступать и заполнят весь диск рано или поздно, в зависимости от длительности паузы, интенсивности записи в WAL и объёма свободного пространства.

9.26.5. Функции синхронизации снимков

Postgres Pro позволяет синхронизировать снимки состояния между сеансами баз данных. *Снимок состояния* определяет, какие данные видны транзакции, работающей с этим снимком. Синхронизация снимков необходима, когда в двух или более сеансах нужно видеть одно и то же содержимое базы данных. Если в двух сеансах транзакции запускаются независимо, всегда есть вероятность, что некая третья транзакция будет зафиксирована между командами `START TRANSACTION` для первых двух, и в результате в одном сеансе будет виден результат третьей, а в другом — нет.

Для решения этой проблемы Postgres Pro позволяет транзакции *экспортировать* снимок состояния, с которым она работает. Пока экспортирующая этот снимок транзакция выполняется, другие транзакции могут *импортировать* его и, таким образом, увидеть абсолютно то же состояние базы данных, что видит первая транзакция. Но учтите, что любые изменения, произведённые этими транзакциями, будут не видны для других, как это и должно быть с изменениями в незафиксированных транзакциях. Таким образом, транзакции синхронизируют только начальное состояние данных, а последующие производимые в них изменения изолируются как обычно.

Снимки состояния экспортируются с помощью функции `pg_export_snapshot`, показанной в Таблице 9.82, и импортируются командой SET TRANSACTION.

Таблица 9.82. Функции синхронизации снимков

Имя	Тип результата	Описание
`pg_export_snapshot()`	`text`	Сохраняет снимок текущего состояния и возвращает его идентификатор

Функция pg_export_snapshot создаёт снимок текущего состояния и возвращает его идентификатор в строке типа text. Данная строка должна передаваться (за рамками базы данных) клиентам, которые будут импортировать этот снимок. При этом импортировать его нужно раньше, чем завершится транзакция, которая его экспортировала. Если необходимо, транзакция может экспортировать несколько снимков. Заметьте, что это имеет смысл только для транзакций уровня READ COMMITTED, так как транзакции REPEATABLE READ и более высоких уровней изоляции работают с одним снимком состояния. После того как транзакция экспортировала снимок, её нельзя подготовить с помощью PREPARE TRANSACTION.

Подробнее использование экспортированных снимков рассматривается в описании SET TRANSACTION.

9.26.6. Функции репликации

В Таблице 9.83 перечислены функции, предназначенные для управления механизмом репликации и взаимодействия с ним. Чтобы изучить этот механизм детальнее, обратитесь к Подразделу 25.2.5, Подразделу 25.2.6 и Главе 47. Использовать эти функции для источников репликации разрешено только суперпользователям, а для слотов репликации — только суперпользователям и пользователям, имеющим право REPLICATION.

Многие из этих функций соответствуют командам в протоколе репликации; см. Раздел 50.4.

Функции, описанные в Подразделе 9.26.3, Подразделе 9.26.4 и Подразделе 9.26.5, также имеют отношение к репликации.

Таблица 9.83. Функции репликации SQL

Функция	Тип результата	Описание
pg_create_physical_replication_slot(*slot_name* name [, *immediately_reserve* boolean, *temporary* boolean])	(*slot_name* name, *lsn* pg_lsn)	Создаёт новый физический слот репликации с именем *slot_name* . Необязательный второй параметр, когда он равен true, указывает, что LSN для этого слота репликации должен быть зарезервирован немедленно; в противном случае LSN резервируется при первом подключении клиента потоковой репликации. Передача изменений из физического слота возможна только по протоколу потоковой репликации — см. Раздел 50.4. Необязательный третий параметр, *temporary*, когда он равен true, указывает, что этот слот не должен постоянно храниться на диске, так как он предназначен только для текущего сеанса. Временные слоты также освобождаются при любой ошибке. Эта функция соответствует команде протокола репликации CREATE_REPLICATION_SLOT ... PHYSICAL.
pg_drop_replication_slot(*slot_name* name)	void	Удаляет физический или логический слот репликации

Функция	**Тип результата**	**Описание**
		с именем *slot_name* . Соответствует команде протокола репликации DROP_REPLICATION_SLOT . Для логических слотов эта функция должна вызываться через подключение к той же базе данных, в которой был создан слот.
pg_create_logical_replication_slot(*slot_name* name, *plugin* name [, *temporary* boolean])	(*slot_name* name, *lsn* pg_lsn)	Создаёт новый логический (декодирующий) слот репликации с именем *slot_name* , используя модуль вывода *plugin*. Необязательный третий параметр, *temporary*, когда равен true, указывает, что этот слот не должен постоянно храниться на диске, так как он предназначен только для текущего сеанса. Временные слоты также освобождаются при любой ошибке. Вызов этой функции равнозначен выполнению команды протокола репликации CREATE_REPLICATION_SLOT ... LOGICAL.
pg_logical_slot_get_changes(*slot_name* name, *upto_lsn* pg_lsn , *upto_nchanges* int, VARIADIC *options* text[])	(*lsn* pg_lsn , *xid* xid, *data* text)	Возвращает изменения в слоте *slot_name* с позиции, до которой ранее были получены изменения. Если параметры *upto_lsn* и *upto_nchanges* равны NULL, логическое декодирование продолжится до конца журнала транзакций. Если *upto_lsn* не NULL, декодироваться будут только транзакции, зафиксированные до заданного LSN. Если *upto_nchanges* не NULL, декодирование остановится, когда число строк, полученных при декодировании, превысит заданное значение. Заметьте однако, что фактическое число возвращённых строк может быть больше, так как это ограничение проверяется только после добавления строк, декодированных для очередной транзакции.
pg_logical_slot_peek_changes(*slot_name* name, *upto_lsn* pg_lsn , *upto_*	(*lsn* pg_lsn , *xid* xid, *data* text)	Работает так же, как функция pg_logical_slot_get_changes() , но не забирает изменения; то есть,

Функция	Тип результата	Описание
nchanges int, VARIADIC *options* text[])		они будут получены снова при следующих вызовах.
pg_logical_slot_get_binary_changes(*slot_name* name, *upto_lsn* pg_lsn , *upto_nchanges* int, VARIADIC *options* text[])	(*lsn* pg_lsn , *xid* xid, *data* bytea)	Работает так же, как функция pg_logical_slot_get_changes() , но выдаёт изменения в типе bytea.
pg_logical_slot_peek_binary_changes(*slot_name* name, *upto_lsn* pg_lsn , *upto_nchanges* int, VARIADIC *options* text[])	(*lsn* pg_lsn , *xid* xid, *data* bytea)	Работает так же, как функция pg_logical_slot_get_changes() , но выдаёт изменения в типе bytea и не забирает их; то есть, они будут получены снова при следующих вызовах.
pg_replication_origin_create(*node_name* text)	oid	Создаёт источник репликации с заданным внешним именем и возвращает назначенный ему внутренний идентификатор.
pg_replication_origin_drop(*node_name* text)	void	Удаляет ранее созданный источник репликации, в том числе связанную информацию о воспроизведении.
pg_replication_origin_oid(*node_name* text)	oid	Ищет источник репликации по имени и возвращает внутренний идентификатор. Если такой источник не находится, возвращает NULL.
pg_replication_origin_session_setup(*node_name* text)	void	Помечает текущий сеанс, как воспроизводящий журнал из указанного источника, что позволяет отслеживать положение воспроизведения. Чтобы отменить это действие, вызовите pg_replication_origin_session_reset . Может использоваться, только если не был настроен предыдущий источник.
pg_replication_origin_session_reset()	void	Отменяет действие pg_replication_origin_session_setup() .
pg_replication_origin_session_is_setup()	bool	В текущем сеансе настроен источник репликации?
pg_replication_origin_session_progress(*flush* bool)	pg_lsn	Возвращает позицию воспроизведения для источника репликации, настроенного в текущем сеансе. Параметр *flush* определяет, будет ли гарантироваться сохранение локальной транзакции на диске.

Функция	Тип результата	Описание
pg_replication_origin_xact_setup(*origin_lsn* pg_lsn , *origin_timestamp* timestamptz)	void	Помечает текущую транзакцию как воспроизводящую транзакцию, зафиксированную с указанным LSN и временем. Может вызываться только после того, как был настроен источник репликации в результате вызова pg_replication_origin_session_setup() .
pg_replication_origin_xact_reset()	void	Отменяет действие pg_replication_origin_xact_setup() .
pg_replication_origin_advance (*node_name* text, *lsn* pg_lsn)	void	Устанавливает положение репликации для заданного узла в указанную позицию. Это в основном полезно для установки начальной позиции или новой позиции после изменения конфигурации и подобных действий. Но учтите, что неосторожное использование этой функции может привести к несогласованности реплицированных данных.
pg_replication_origin_progress(*node_name* text, *flush* bool)	pg_lsn	Возвращает позицию воспроизведения для заданного источника репликации. Параметр *flush* определяет, будет ли гарантироваться сохранение локальной транзакции на диске.
pg_logical_emit_message(*transactional* bool, *prefix* text, *content* text)	pg_lsn	Выдаёт текстовое сообщение логического декодирования. Её можно использовать для передачи произвольных сообщений через WAL модулям логического декодирования. Параметр *transactional* устанавливает, должно ли сообщение быть частью текущей транзакции или оно должно записываться немедленно и декодироваться сразу, как только эта запись будет прочитана при логическом декодировании. Параметр *prefix* задаёт текстовый префикс, по которому модули логического декодирования могут легко распознать интересующие их сообщения. В параметре

Функция	Тип результата	Описание
		content передаётся текст сообщения.
pg_logical_emit_message(*transactional* bool, *prefix* text, *content* bytea)	pg_lsn	Выдаёт двоичное сообщение логического декодирования. Её можно использовать для передачи произвольных сообщений через WAL модулям логического декодирования. Параметр *transactional* устанавливает, должно ли сообщение быть частью текущей транзакции или оно должно записываться немедленно и декодироваться сразу, как только эта запись будет прочитана при логическом декодировании. Параметр *prefix* задаёт текстовый префикс, по которому модули логического декодирования могут легко распознать интересующие их сообщения. В параметре *content* передаётся двоичное содержимое сообщения.

9.26.7. Функции управления объектами баз данных

Функции, перечисленные в Таблице 9.84, вычисляют объём, который занимают на диске различные объекты баз данных.

Таблица 9.84. Функции получения размера объектов БД

Имя	Тип результата	Описание
pg_column_size(any)	int	Число байт, необходимых для хранения заданного значения (возможно, в сжатом виде)
pg_database_size(oid)	bigint	Объём, который занимает на диске база данных с заданным OID
pg_database_size(name)	bigint	Объём, который занимает на диске база данных с заданным именем
pg_indexes_size(regclass)	bigint	Общий объём индексов, связанных с указанной таблицей
pg_relation_size(*relation* regclass, *fork* text)	bigint	Объём, который занимает на диске указанный слой ('main', 'fsm', 'vm' или 'init') заданной таблицы или индекса
pg_relation_size(*relation* regclass)	bigint	Краткая форма pg_relation_size(..., 'main')

Имя	Тип результата	Описание
pg_size_bytes(text)	bigint	Преобразует размер в понятном человеку формате с единицами измерения в число байт
pg_size_pretty(bigint)	text	Преобразует размер в байтах, представленный в 64-битном целом, в понятный человеку формат с единицами измерения
pg_size_pretty(numeric)	text	Преобразует размер в байтах, представленный в значении числового типа, в понятный человеку формат с единицами измерения
pg_table_size(regclass)	bigint	Объём, который занимает на диске данная таблица, за исключением индексов (но включая TOAST, карту свободного места и карту видимости)
pg_tablespace_size(oid)	bigint	Объём, который занимает на диске табличное пространство с указанным OID
pg_tablespace_size(name)	bigint	Объём, который занимает на диске табличное пространство с заданным именем
pg_total_relation_size(regclass)	bigint	Общий объём, который занимает на диске заданная таблица, включая все индексы и данные TOAST

pg_column_size показывает, какой объём требуется для хранения данного значения.

pg_total_relation_size принимает OID или имя таблицы или данных TOAST и возвращает общий объём, который занимает на диске эта таблица, включая все связанные с ней индексы. Результат этой функции равняется pg_table_size + pg_indexes_size.

pg_table_size принимает OID или имя таблицы и возвращает объём, который занимает на диске эта таблица без индексов. (При этом учитывается размер TOAST, карты свободного места и карты видимости.)

pg_indexes_size принимает OID или имя таблицы и возвращает общий объём, который занимают все индексы таблицы.

pg_database_size и pg_tablespace_size принимают OID или имя базы данных либо табличного пространства и возвращают общий объём, который они занимают на диске. Для использования pg_database_size требуется иметь право CONNECT для указанной базы данных (оно предоставляется по умолчанию) или быть членом роли pg_read_all_stats. Для использования pg_tablespace_size необходимо иметь право CREATE для указанного табличного пространства или быть членом роли pg_read_all_stats, если только это не табличное пространство по умолчанию для текущей базы данных.

pg_relation_size принимает OID или имя таблицы, индекса или TOAST-таблицы и возвращает размер одного слоя этого отношения (в байтах). (Заметьте, что в большинстве случаев удобнее использовать более высокоуровневые функции pg_total_relation_size и pg_table_size, которые суммируют размер всех слоёв.) С одним аргументом она возвращает размер основного слоя для

данных заданного отношения. Название другого интересующего слоя можно передать во втором аргументе:

- `'main'` возвращает размер основного слоя данных заданного отношения.
- `'fsm'` возвращает размер карты свободного места (см. Раздел 63.3), связанной с заданным отношением.
- `'vm'` возвращает размер карты видимости (см. Раздел 63.4), связанной с заданным отношением.
- `'init'` возвращает размер слоя инициализации для заданного отношения, если он имеется.

`pg_size_pretty` можно использовать для форматирования результатов других функций в виде, более понятном человеку, с единицами измерения bytes, kB, MB, GB и TB.

`pg_size_bytes` можно использовать для получения размера в байтах из строки в формате, понятном человеку. Входная строка может содержать единицы bytes, kB, MB, GB и TB, и разбирается без учёта регистра. Если единицы не указываются, подразумеваются байты.

> **Примечание**
>
> Единицы kB, MB, GB и TB, фигурирующие в функциях `pg_size_pretty` и `pg_size_bytes`, определяются как степени 2, а не 10, так что 1kB равен 1024 байтам, 1MB равен 1024^2 = 1048576 байт и т. д.

Вышеописанные функции, работающие с таблицами или индексами, принимают аргумент типа `regclass`, который представляет собой просто OID таблицы или индекса в системном каталоге `pg_class`. Однако вам не нужно вручную вычислять OID, так как процедура ввода значения `regclass` может сделать это за вас. Для этого достаточно записать имя таблицы в апострофах, как обычную текстовую константу. В соответствии с правилами обработки обычных имён SQL, если имя таблицы не заключено в кавычки, эта строка будет переведена в нижний регистр.

Если переданному значению OID не соответствуют существующий объект, эти функции возвращают NULL.

Функции, перечисленные в Таблице 9.85, помогают определить, в каких файлах на диске хранятся объекты базы данных.

Таблица 9.85. Функции определения расположения объектов

Имя	Тип результата	Описание
`pg_relation_filenode(` *relation* `regclass)`	`oid`	Номер файлового узла для указанного отношения
`pg_relation_filepath(` *relation* `regclass)`	`text`	Путь к файлу, в котором хранится указанное отношение
`pg_filenode_relation(` *tablespace* `oid,` *filenode* `oid)`	`regclass`	Находит отношение, связанное с данным табличным пространством и файловым узлом

`pg_relation_filenode` принимает OID или имя таблицы, индекса, последовательности или таблицы TOAST и возвращает номер «файлового узла», связанным с этим объектом. Файловым узлом называется основной компонент имени файла, используемого для хранения данных (подробнее это описано в Разделе 63.1). Для большинства таблиц этот номер совпадает со значением `pg_class.relfilenode`, но для некоторых системных каталогов `relfilenode` равен 0, и нужно использовать эту функцию, чтобы узнать действительное значение. Если указанное отношение не хранится на диске, как например представление, данная функция возвращает NULL.

`pg_relation_filepath` подобна `pg_relation_filenode`, но возвращает полный путь к файлу (относительно каталога данных `PGDATA`) отношения.

Функция pg_filenode_relation является обратной к pg_relation_filenode. Она возвращает OID отношения по заданному OID «табличного пространства» и «файловому узлу». Для таблицы в табличном пространстве по умолчанию в первом параметре можно передать 0.

В Таблица 9.86 перечислены функции, предназначенные для управления правилами сортировки.

Таблица 9.86. Функции управления правилами сортировки

Имя	Тип результата	Описание
pg_collation_actual_version(oid)	text	выдаёт действующую версию правила сортировки из операционной системы
pg_import_system_collations(*schema* regnamespace)	integer	импортирует правила сортировки из операционной системы

Функция pg_collation_actual_version возвращает действующую версию объекта правила сортировки, которая в настоящее время установлена в операционной системе. Если она отличается от значения в pg_collation.collversion, может потребоваться перестроить объекты, зависящие от данного правила сортировки. См. также ALTER COLLATION.

Функция pg_import_system_collations добавляет правила сортировки в системный каталог pg_collation, анализируя все локали, которые она находит в операционной системе. Эту информацию использует initdb; за подробностями обратитесь к Подразделу 22.2.2. Если позднее в систему устанавливаются дополнительные локали, эту функцию можно запустить снова, чтобы добавились правила сортировки для новых локалей. Локали, для которых обнаруживаются существующие записи в pg_collation, будут пропущены. (Объекты правил сортировки для локалей, которые перестают существовать в операционной системе, никогда не удаляются этой функцией.) В параметре *schema* обычно передаётся pg_catalog, но это не обязательно; правила сортировки могут быть установлены и в другую схему. Эта функция возвращает число созданных ей объектов правил сортировки; вызывать её разрешено только суперпользователям.

9.26.8. Функции обслуживания индексов

В Таблице 9.87 перечислены функции, предназначенные для обслуживания индексов. Эти функции нельзя вызывать в процессе восстановления. Использовать эти функции разрешено только суперпользователям и владельцу определённого индекса.

Таблица 9.87. Функции обслуживания индексов

Имя	Тип результата	Описание
brin_summarize_new_values(*index* regclass)	integer	обобщает ещё не обобщённые зоны страниц
brin_summarize_range(*index* regclass, *blockNumber* bigint)	integer	обобщает зону страниц, охватывающую данный блок (если она ещё не обобщена)
brin_desummarize_range(*index* regclass, *blockNumber* bigint)	integer	сброс обобщения зоны страниц, охватывающей данный блок (если она была обобщена)
gin_clean_pending_list(*index* regclass)	bigint	перемещает элементы из списка записей GIN, ожидающих обработки, в основную структуру индекса

Функция brin_summarize_new_values принимает OID или имя индекса BRIN и просматривает индекс в поисках зон страниц в базовой таблице, ещё не обобщённых в индексе; для каждой такой

зоны в результате сканирования страниц таблицы создаётся новый обобщающий кортеж в индексе. Возвращает эта функция число вставленных в индекс обобщающих записей о зонах страниц. Функция `brin_summarize_range` делает то же самое, но затрагивает только зону, охватывающую блок с заданным номером.

Функция `gin_clean_pending_list` принимает OID или имя индекса GIN и очищает очередь указанного индекса, массово перемещая элементы из неё в основную структуру данных GIN. Возвращает она число страниц, убранных из очереди. Заметьте, что если для обработки ей передаётся индекс GIN, построенный с отключённым параметром `fastupdate`, очистка не производится и возвращается значение 0, так как у такого индекса нет очереди записей. Подробнее об очереди записей и параметре `fastupdate` рассказывается в Подразделе 60.4.1 и Разделе 60.5.

9.26.9. Функции для работы с обычными файлами

Функции, перечисленные в Таблице 9.88, предоставляют прямой доступ к файлам, находящимся на сервере. Они позволяют обращаться только к файлам в каталоге кластера баз данных (по относительному пути) или в каталоге `log_directory` (по пути, заданному в параметре конфигурации `log_directory`). Использовать эти функции могут только суперпользователи, если явно не отмечено иное.

Таблица 9.88. Функции для работы с обычными файлами

Имя	Тип результата	Описание
`pg_ls_dir( *dirname* text [, *missing_ok* boolean, *include_dot_dirs* boolean])`	`setof text`	Возвращает список содержимого каталога.
`pg_ls_logdir()`	`setof record`	Выводит имя, размер и время последнего изменения файлов в каталоге журнала сообщений. Доступ к этой функции имеют члены роли `pg_monitor` , также его можно разрешить и другим ролям обычных пользователей.
`pg_ls_waldir()`	`setof record`	Выводит имя, размер и время последнего изменения файлов в каталоге WAL. Доступ к этой функции имеют члены роли `pg_monitor` , также его можно разрешить и другим ролям обычных пользователей.
`pg_read_file( *filename* text [, *offset* bigint, *length* bigint [, *missing_ok* boolean] ])`	`text`	Возвращает содержимое текстового файла.
`pg_read_binary_file( *filename* text [, *offset* bigint, *length* bigint [, *missing_ok* boolean] ])`	`bytea`	Возвращает содержимое файла.
`pg_stat_file( *filename* text[, *missing_ok* boolean])`	`record`	Возвращает информацию о файле.

Некоторые из этих функций принимают необязательный параметр *missing_ok*, который определяет их поведение в случае отсутствия файла или каталога. Если он равен `true`, функция

возвращает NULL (за исключением pg_ls_dir, которая возвращает пустое множество). Если он равен false, возникает ошибка. Значение по умолчанию — false.

pg_ls_dir возвращает имена всех файлов (а также каталогов и других специальных файлов) в заданном каталоге. Параметр *include_dot_dirs* определяет, будут ли в результирующий набор включаться каталоги «.» и «..». По умолчанию они не включаются (false), но их можно включить, чтобы с параметром *missing_ok* равным true, пустой каталог можно было отличить от несуществующего.

pg_ls_logdir возвращает имя, размер и время последнего изменения (mtime) всех файлов в каталоге журналов сообщений. По умолчанию использовать её разрешено только суперпользователям и членам роли pg_monitor. Другим пользователям доступ к ней можно дать, используя GRANT.

pg_ls_waldir возвращает имя, размер и время последнего изменения (mtime) всех файлов в каталоге журнала предзаписи (WAL). По умолчанию использовать её разрешено только суперпользователям и членам роли pg_monitor. Другим пользователям доступ к ней можно дать, используя GRANT.

pg_read_file возвращает фрагмент текстового файла с заданного смещения (*offset*), размером не больше *length* байт (размер может быть меньше, если файл кончится раньше). Если смещение *offset* отрицательно, оно отсчитывается от конца файла. Если параметры *offset* и *length* опущены, возвращается всё содержимое файла. Прочитанные из файла байты обрабатываются как символы в серверной кодировке; если они оказываются недопустимыми для этой кодировки, возникает ошибка.

pg_read_binary_file подобна pg_read_file, но её результат имеет тип bytea; как следствие, никакие проверки кодировки не выполняются. В сочетании с convert_from эту функцию можно применять для чтения файлов в произвольной кодировке:

```
SELECT convert_from(pg_read_binary_file('file_in_utf8.txt'), 'UTF8');
```

pg_stat_file возвращает запись, содержащую размер файла, время последнего обращения и последнего изменения, а также время последнего изменения состояния (только в Unix-системах), время создания (только в Windows) и признак типа boolean, показывающий, что это каталог. Примеры использования:

```
SELECT * FROM pg_stat_file('filename');
SELECT (pg_stat_file('filename')).modification;
```

9.26.10. Функции управления рекомендательными блокировками

Функции, перечисленные в Таблице 9.89, предназначены для управления рекомендательными блокировками. Подробнее об их использовании можно узнать в Подразделе 13.3.5.

Таблица 9.89. Функции управления рекомендательными блокировками

Имя	Тип результата	Описание
pg_advisory_lock(*key* bigint)	void	Получает исключительную блокировку на уровне сеанса
pg_advisory_lock(*key1* int, *key2* int)	void	Получает исключительную блокировку на уровне сеанса
pg_advisory_lock_shared(*key* bigint)	void	Получает разделяемую блокировку на уровне сеанса
pg_advisory_lock_shared(*key1* int, *key2* int)	void	Получает разделяемую блокировку на уровне сеанса

Имя	Тип результата	Описание
pg_advisory_unlock(*key* bigint)	boolean	Освобождает исключительную блокировку на уровне сеанса
pg_advisory_unlock(*key1* int, *key2* int)	boolean	Освобождает исключительную блокировку на уровне сеанса
pg_advisory_unlock_all()	void	Освобождает все блокировки на уровне сеанса, удерживаемые в данном сеансе
pg_advisory_unlock_ shared(*key* bigint)	boolean	Освобождает разделяемую блокировку на уровне сеанса
pg_advisory_unlock_ shared(*key1* int, *key2* int)	boolean	Освобождает разделяемую блокировку на уровне сеанса
pg_advisory_xact_lock(*key* bigint)	void	Получает исключительную блокировку на уровне транзакции
pg_advisory_xact_lock(*key1* int, *key2* int)	void	Получает исключительную блокировку на уровне транзакции
pg_advisory_xact_lock_ shared(*key* bigint)	void	Получает разделяемую блокировку на уровне транзакции
pg_advisory_xact_lock_ shared(*key1* int, *key2* int)	void	Получает разделяемую блокировку на уровне транзакции
pg_try_advisory_lock(*key* bigint)	boolean	Получает исключительную блокировку на уровне сеанса, если это возможно
pg_try_advisory_lock(*key1* int, *key2* int)	boolean	Получает исключительную блокировку на уровне сеанса, если это возможно
pg_try_advisory_lock_ shared(*key* bigint)	boolean	Получает разделяемую блокировку на уровне сеанса, если это возможно
pg_try_advisory_lock_ shared(*key1* int, *key2* int)	boolean	Получает разделяемую блокировку на уровне сеанса, если это возможно
pg_try_advisory_xact_ lock(*key* bigint)	boolean	Получает исключительную блокировку на уровне транзакции, если это возможно
pg_try_advisory_xact_ lock(*key1* int, *key2* int)	boolean	Получает исключительную блокировку на уровне транзакции, если это возможно
pg_try_advisory_xact_ lock_shared(*key* bigint)	boolean	Получает разделяемую блокировку на уровне транзакции, если это возможно
pg_try_advisory_xact_ lock_shared(*key1* int, *key2* int)	boolean	Получает разделяемую блокировку на уровне транзакции, если это возможно

pg_advisory_lock блокирует определённый приложением ресурс, задаваемый одним 64-битным или двумя 32-битными ключами (заметьте, что их значения не пересекаются). Если

идентификатор этого ресурса удерживает другой сеанс, эта функция не завершится, пока ресурс не станет доступным. Данная функция устанавливает блокировку в исключительном режиме. Если поступает сразу несколько запросов на блокировку, они накапливаются, так что если один ресурс был заблокирован три раза, его необходимо три раза разблокировать, чтобы он был доступен в других сеансах.

`pg_advisory_lock_shared` работает подобно `pg_advisory_lock`, но позволяет разделять блокировку с другими сеансами, запрашивающими её как разделяемую. Выполнение может быть приостановлено, только если другой сеанс запросил её в исключительном режиме.

`pg_try_advisory_lock` работает подобно `pg_advisory_lock`, но не ждёт освобождения ресурса. Эта функция либо немедленно получает блокировку и возвращает `true`, либо сразу возвращает `false`, если получить её не удаётся.

`pg_try_advisory_lock_shared` работает как `pg_try_advisory_lock`, но пытается получить разделяемую, а не исключительную блокировку.

`pg_advisory_unlock` освобождает ранее полученную исключительную блокировку на уровне сеанса. Если блокировка освобождена успешно, эта функция возвращает `true`, а если она не была занята — `false`, при этом сервер выдаёт предупреждение SQL.

`pg_advisory_unlock_shared` работает подобно `pg_advisory_unlock`, но освобождает разделяемую блокировку на уровне сеанса.

`pg_advisory_unlock_all` освобождает все блокировки на уровне сеанса, закреплённые за текущим сеансом. (Эта функция неявно вызывается в конце любого сеанса, даже при нештатном отключении клиента.)

`pg_advisory_xact_lock` работает подобно `pg_advisory_lock`, но её блокировка автоматически освобождается в конце текущей транзакции и не может быть освобождена явным образом.

`pg_advisory_xact_lock_shared` подобна функции `pg_advisory_lock_shared`, но её блокировка автоматически освобождается в конце текущей транзакции и не может быть освобождена явным образом.

`pg_try_advisory_xact_lock` работает подобно `pg_try_advisory_lock`, но её блокировка (если она была получена) автоматически освобождается в конце текущей транзакции и не может быть освобождена явным образом.

`pg_try_advisory_xact_lock_shared` работает подобно `pg_try_advisory_lock_shared`, но её блокировка (если она была получена) автоматически освобождается в конце текущей транзакции и не может быть освобождена явным образом.

9.26.11. Отладочные функции

Функция, показанная в Таблице 9.90, может быть полезна для низкоуровневых операций, например в целях отладки или исследования испорченных баз данных Postgres Pro.

Таблица 9.90. Функции синхронизации снимков

Имя	Тип результата	Описание
`pg_snapshot_any()`	`void`	Отключает действие правил MVCC в рамках текущей

Имя	Тип результата	Описание
		транзакции, что позволяет видеть в ней все версии данных.

Применяйте `pg_snapshot_any` крайне осторожно. Выполнять её следует в транзакции с уровнем изоляции `REPEATABLE READ` или выше, в противном случае следующий же запрос заменит выбранный снимок новым. Выполнять её могут только суперпользователи.

> **Примечание**
>
> Если ваш кластер БД создавался сервером версии, в которой этой функции ещё не было, определите её следующей командой:
>
> ```
> CREATE FUNCTION pg_snapshot_any() RETURNS void AS 'pg_snapshot_any' LANGUAGE
>  internal;
> ```

9.27. Триггерные функции

В настоящее время в Postgres Pro есть единственная встроенная триггерная функция, `suppress_redundant_updates_trigger`, которая предотвращает изменения, фактически не влияющие на данные в строке, тогда как обычно изменения выполняются вне зависимости от того, были ли изменены данные. (Обычное поведение не предполагает сравнения данных, поэтому изменения выполняются быстрее, и в ряде случаев именно это поведение желательно.)

В идеале, следует избегать операций изменения, которые фактически не меняют данные в записях. Подобные ненужные изменения могут обходиться дорого, особенно когда требуется обновлять множество индексов, к тому же впоследствии базу данных придётся очищать от «мёртвых» строк. Однако выявить такие изменения в клиентском коде бывает сложно, если вообще возможно, а при составлении соответствующих проверочных выражений легко допустить ошибку. В качестве альтернативного решения можно использовать функцию `suppress_redundant_updates_trigger`, которая опускает изменения, не меняющие данные. Однако использовать её следует с осторожностью. Данный триггер выполняется не большое, но значительное время для каждой записи, так что если большинство записей всё-таки фактически изменяются, этот триггер скорее замедлит операцию изменения.

Функцию `suppress_redundant_updates_trigger` можно привязать к таблице так:

```
CREATE TRIGGER z_min_update
BEFORE UPDATE ON tablename
FOR EACH ROW EXECUTE PROCEDURE suppress_redundant_updates_trigger();
```

В большинстве случаев этот триггер должен вызываться для каждой строки последним. А так как триггеры вызываются по порядку сортировки их имён, имя для него нужно выбирать таким, чтобы оно было последним среди имён всех триггеров, которые могут быть в таблице.

Подробнее о создании триггеров можно узнать в описании CREATE TRIGGER.

9.28. Функции событийных триггеров

Postgres Pro предоставляет следующие вспомогательные функции для получения информации в событийных триггерах.

Подробнее о событийных триггерах можно узнать в Главе 37.

9.28.1. Получение изменений в конце команды

Функция `pg_event_trigger_ddl_commands` возвращает список команд DDL, выполняемых в результате действия пользователя. Вызывать её можно только в функции, реализующей

событийный триггер ddl_command_end. При попытке вызвать её в любом другом контексте возникнет ошибка. Функция pg_event_trigger_ddl_commands возвращает одну строку для каждой базовой команды; для некоторых команд, записываемых в виде одного предложения SQL, может возвращаться несколько строк. Эта функция возвращает следующие столбцы:

Имя	Тип	Описание
classid	oid	OID каталога, к которому относится объект
objid	oid	OID самого объекта
objsubid	integer	Идентификатор подобъекта (например, номер для столбца)
command_tag	text	Тег команды
object_type	text	Тип объекта
schema_name	text	Имя схемы, к которой относится объект; если объект не относится ни к какой схеме — NULL. В кавычки имя не заключается.
object_identity	text	Текстовое представление идентификатора объекта, включающее схему. При необходимости компоненты этого идентификатора заключаются в кавычки.
in_extension	bool	True, если команда является частью скрипта расширения
command	pg_ddl_command	Полное представление команды, во внутреннем формате. Его нельзя вывести непосредственно, но можно передать другим функциям, чтобы получить различные сведения о команде.

9.28.2. Обработка объектов, удалённых командой DDL

Функция pg_event_trigger_dropped_objects возвращает список всех объектов, удалённых командой, вызвавшей событие sql_drop. При вызове в другом контексте pg_event_trigger_dropped_objects выдаёт ошибку. pg_event_trigger_dropped_objects возвращает следующие столбцы:

Имя	Тип	Описание
classid	oid	OID каталога, к которому относился объект
objid	oid	OID самого объекта
objsubid	integer	Идентификатор подобъекта (например, номер для столбца)
original	bool	True, если это один из корневых удаляемых объектов
normal	bool	True, если к этому объекту в графе зависимостей

Имя	Тип	Описание
		привело отношение обычной зависимости
is_temporary	bool	True, если объект был временным
object_type	text	Тип объекта
schema_name	text	Имя схемы, к которой относился объект; если объект не относился ни к какой схеме — NULL. В кавычки имя не заключается.
object_name	text	Имя объекта, если сочетание схемы и имени позволяет уникально идентифицировать объект; в противном случае — NULL. Имя не заключается в кавычки и не дополняется именем схемы.
object_identity	text	Текстовое представление идентификатора объекта, включающее схему. При необходимости компоненты этого идентификатора заключаются в кавычки.
address_names	text[]	Массив, который в сочетании с object_type и массивом address_args можно передать функции pg_get_object_address() , чтобы воссоздать адрес объекта на удалённом сервере, содержащем одноимённый объект того же рода.
address_args	text[]	Дополнение к массиву address_names

Функцию pg_event_trigger_dropped_objects можно использовать в событийном триггере так:

```
CREATE FUNCTION test_event_trigger_for_drops()
        RETURNS event_trigger LANGUAGE plpgsql AS $$
DECLARE
    obj record;
BEGIN
    FOR obj IN SELECT * FROM pg_event_trigger_dropped_objects()
    LOOP
        RAISE NOTICE '% dropped object: % %.% %',
                     tg_tag,
                     obj.object_type,
                     obj.schema_name,
                     obj.object_name,
                     obj.object_identity;
    END LOOP;
END;
$$;
CREATE EVENT TRIGGER test_event_trigger_for_drops
```

```
   ON sql_drop
   EXECUTE PROCEDURE test_event_trigger_for_drops();
```

9.28.3. Обработка события перезаписи таблицы

В Таблице 9.91 показаны функции, выдающие информацию о таблице, для которой произошло событие перезаписи таблицы (`table_rewrite`). При попытке вызвать их в другом контексте возникнет ошибка.

Таблица 9.91. Информация о перезаписи таблицы

Имя	Тип результата	Описание
`pg_event_trigger_table_rewrite_oid()`	`Oid`	OID таблицы, которая будет перезаписана.
`pg_event_trigger_table_rewrite_reason()`	`int`	Код причины, показывающий, чем вызвана перезапись. Точное значение кодов зависит от выпуска (версии).

Функцию `pg_event_trigger_table_rewrite_oid` можно использовать в событийном триггере так:

```
CREATE FUNCTION test_event_trigger_table_rewrite_oid()
 RETURNS event_trigger
 LANGUAGE plpgsql AS
$$
BEGIN
  RAISE NOTICE 'rewriting table % for reason %',
                pg_event_trigger_table_rewrite_oid()::regclass,
                pg_event_trigger_table_rewrite_reason();
END;
$$;

CREATE EVENT TRIGGER test_table_rewrite_oid
                  ON table_rewrite
   EXECUTE PROCEDURE test_event_trigger_table_rewrite_oid();
```

Глава 10. Преобразование типов

SQL-операторы, намеренно или нет, требуют совмещать данные разных типов в одном выражении. Для вычисления подобных выражений со смешанными типами Postgres Pro предоставляет широкий набор возможностей.

Очень часто пользователю не нужно понимать все тонкости механизма преобразования. Однако следует учитывать, что неявные преобразования, производимые Postgres Pro, могут влиять на результат запроса. Поэтому при необходимости нужные результаты можно получить, применив *явное* преобразование типов.

В этой главе описываются общие механизмы преобразования типов и соглашения, принятые в Postgres Pro. За дополнительной информацией о конкретных типах данных и разрешённых для них функциях и операторах обратитесь к соответствующим разделам в Главе 8 и Главе 9.

10.1. Обзор

SQL — язык со строгой типизацией. То есть каждый элемент данных в нём имеет некоторый тип, определяющий его поведение и допустимое использование. Postgres Pro наделён расширяемой системой типов, более универсальной и гибкой по сравнению с другими реализациями SQL. При этом преобразования типов в Postgres Pro в основном подчиняются определённым общим правилам, для их понимания не нужен *эвристический* анализ. Благодаря этому в выражениях со смешанными типами можно использовать даже типы, определённые пользователями.

Анализатор выражений Postgres Pro разделяет их лексические элементы на пять основных категорий: целые числа, другие числовые значения, текстовые строки, идентификаторы и ключевые слова. Константы большинства не числовых типов сначала классифицируются как строки. В определении языка SQL допускается указывать имена типов в строках и это можно использовать в Postgres Pro, чтобы направить анализатор по верному пути. Например, запрос:

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";

 label  | value
--------+-------
 Origin | (0,0)
(1 row)
```

содержит две строковых константы, типа `text` и типа `point`. Если для такой константы не указан тип, для неё первоначально предполагается тип `unknown`, который затем может быть уточнён, как описано ниже.

В SQL есть четыре фундаментальных фактора, определяющих правила преобразования типов для анализатора выражений Postgres Pro:

Вызовы функций

Система типов Postgres Pro во многом построена как дополнение к богатым возможностям функций. Функции могут иметь один или несколько аргументов, и при этом Postgres Pro разрешает перегружать имена функций, так что имя функции само по себе не идентифицирует вызываемую функцию; анализатор выбирает правильную функцию в зависимости от типов переданных аргументов.

Операторы

Postgres Pro позволяет использовать в выражениях префиксные и постфиксные операторы с одним аргументом, а также операторы с двумя аргументами. Как и функции, операторы можно перегружать, так что и с ними существует проблема выбора правильного оператора.

Сохранение значений

SQL-операторы `INSERT` и `UPDATE` помещают результаты выражений в таблицы. При этом получаемые значения должны соответствовать типам целевых столбцов или, возможно, приводиться к ним.

`UNION`, `CASE` и связанные конструкции

Так как все результаты запроса объединяющего оператора `SELECT` должны оказаться в одном наборе столбцов, результаты каждого подзапроса `SELECT` должны приводиться к одному набору типов. Подобным образом, результирующие выражения конструкции `CASE` должны приводиться к общему типу, так как выражение `CASE` в целом должно иметь определённый выходной тип. Подобное определение общего типа для значений нескольких подвыражений требуется и для некоторых других конструкций, например `ARRAY[]`, а также для функций `GREATEST` и `LEAST`.

Информация о существующих преобразованиях или *приведениях* типов, для каких типов они определены и как их выполнять, хранится в системных каталогах. Пользователь также может добавить дополнительные преобразования с помощью команды CREATE CAST. (Обычно это делается, когда определяются новые типы данных. Набор приведений для встроенных типов достаточно хорошо проработан, так что его лучше не менять.)

Дополнительная логика анализа помогает выбрать оптимальное приведение в группах типов, допускающих неявные преобразования. Для этого типы данных разделяются на несколько базовых *категорий*, которые включают: `boolean`, `numeric`, `string`, `bitstring`, `datetime`, `timespan`, `geometric`, `network` и пользовательские типы. (Полный список категорий приведён в Таблице 49.63; хотя его тоже можно расширить, определив свои категории.) В каждой категории могут быть выбраны один или несколько *предпочитаемых типов*, которые будут считаться наиболее подходящими при рассмотрении нескольких вариантов. Аккуратно выбирая предпочитаемые типы и допустимые неявные преобразования, можно добиться того, что выражения с неоднозначностями (в которых возможны разные решения задачи преобразования) будут разрешаться наилучшим образом.

Все правила преобразования типов разработаны с учётом следующих принципов:

- Результат неявных преобразованиях всегда должен быть предсказуемым и понятным.
- Если в неявном преобразовании нет нужды, анализатор и исполнитель запроса не должны тратить лишнее время на это. То есть, если запрос хорошо сформулирован и типы значений совпадают, он должен выполняться без дополнительной обработки в анализаторе и без лишних вызовов неявных преобразований.
- Кроме того, если запрос изначально требовал неявного преобразования для функции, а пользователь определил новую функцию с точно совпадающими типами аргументов, анализатор должен переключиться на новую функцию и больше не выполнять преобразование для вызова старой.

10.2. Операторы

При выборе конкретного оператора, задействованного в выражении, PostgreSQL следует описанному ниже алгоритму. Заметьте, что на этот выбор могут неявно влиять приоритеты остальных операторов в данном выражении, так как они определяют, какие подвыражения будут аргументами операторов. Подробнее об этом рассказывается в Подразделе 4.1.6.

Выбор оператора по типу

1. Выбрать операторы для рассмотрения из системного каталога `pg_operator`. Если имя оператора не дополнено именем схемы (обычно это так), будут рассматриваться все операторы с подходящим именем и числом аргументов, видимые в текущем пути поиска (см. Подраздел 5.8.3). Если имя оператора определено полностью, в рассмотрение принимаются только операторы из указанной схемы.

    - (Optional) Если в пути поиска оказывается несколько операторов с одинаковыми типами аргументов, учитываются только те из них, которые находятся в пути раньше. Операторы с разными типами аргументов рассматриваются на равных правах вне зависимости от их положения в пути поиска.

2. Проверить, нет ли среди них оператора с точно совпадающими типами аргументов. Если такой оператор есть (он может быть только одним в отобранном ранее наборе), использовать его. Отсутствие точного совпадения создаёт угрозу вызова с указанием полного имени [1] (нетипичным) любого оператора, который может оказаться в схеме, где могут создавать объекты недоверенные пользователи. В таких ситуациях приведите типы аргументов для получения точного совпадения.

   a. (Optional) Если один аргумент при вызове бинарного оператора имеет тип `unknown`, для данной проверки предполагается, что он имеет тот же тип, что и второй его аргумент. При вызове бинарного оператора с двумя аргументами `unknown` или унарного с одним `unknown` оператор не будет выбран на этом шаге.

   b. (Optional) Если один аргумент при вызове бинарного оператора имеет тип `unknown`, а другой — домен, проверить, есть ли оператор, принимающий базовый тип домена с обеих сторон; если таковой находится, использовать его.

3. Найти самый подходящий.

   a. Отбросить кандидатов, для которых входные типы не совпадают и не могут быть преобразованы (неявным образом) так, чтобы они совпали. В данном случае считается, что константы типа `unknown` можно преобразовать во что угодно. Если остаётся только один кандидат, использовать его, в противном случае перейти к следующему шагу.

   b. Если один из аргументов имеет тип домен, далее считать его типом базовый тип домена. Благодаря этому при поиске неоднозначно заданного оператора домены будут подобны свои базовым типам.

   c. Просмотреть всех кандидатов и оставить только тех, для которых точно совпадают как можно больше типов аргументов. Оставить всех кандидатов, если точных совпадений нет. Если остаётся только один кандидат, использовать его, в противном случае перейти к следующему шагу.

   d. Просмотреть всех кандидатов и оставить только тех, которые принимают предпочитаемые типы (из категории типов входных значений) в наибольшем числе позиций, где требуется преобразование типов. Оставить всех кандидатов, если ни один не принимает предпочитаемые типы. Если остаётся только один кандидат, использовать его, в противном случае перейти к следующему шагу.

   e. Если какие-либо значения имеют тип `unknown`, проверить категории типов, принимаемых в данных позициях аргументов оставшимися кандидатами. Для каждой позиции выбрать категорию `string`, если какой-либо кандидат принимает эту категорию. (Эта склонность к строкам объясняется тем, что константа типа unknown выглядит как строка.) Если эта категория не подходит, но все оставшиеся кандидаты принимают одну категорию, выбрать её; в противном случае констатировать неудачу — сделать правильный выбор без дополнительных подсказок нельзя. Затем отбросить кандидатов, которые не принимают типы выбранной категории. Далее, если какой-либо кандидат принимает предпочитаемый тип из этой категории, отбросить кандидатов, принимающих другие, не предпочитаемые типы для данного аргумента. Оставить всех кандидатов, если эти проверки не прошёл ни один. Если остаётся только один кандидат, использовать его, в противном случае перейти к следующему шагу.

   f. Если в списке аргументов есть аргументы и типа `unknown`, и известного типа, и этот известный тип один для всех аргументов, предположить, что аргументы типа `unknown` также имеют этот тип, и проверить, какие кандидаты могут принимать этот тип в позиции аргумента `unknown`. Если остаётся только один кандидат, использовать его, в противном случае констатировать неудачу.

Ниже это проиллюстрировано на примерах.

[1] Эта угроза неактуальна для имён без схемы, так как путь поиска, содержащий схемы, в которых недоверенные пользователи могут создавать объекты, не соответствует шаблону безопасного использования схем.

Пример 10.1. Разрешение типа для оператора факториала

В стандартном каталоге определён только один оператор факториала (постфиксный `!`) и он принимает аргумент типа `bigint`. При просмотре следующего выражения его аргументу изначально назначается тип `integer`:

```
SELECT 40 ! AS "40 factorial";

                  40 factorial
-----------------------------------------------
 815915283247897734345611269596115894272000000000
(1 row)
```

Анализатор выполняет преобразование типа для этого операнда и запрос становится равносильным:

```
SELECT CAST(40 AS bigint) ! AS "40 factorial";
```

Пример 10.2. Разрешение оператора конкатенации строк

Синтаксис текстовых строк используется как для записи строковых типов, так и для сложных типов расширений. Если тип не указан явно, такие строки сопоставляются по тому же алгоритму с наиболее подходящими операторами.

Пример с одним неопределённым аргументом:

```
SELECT text 'abc' || 'def' AS "text and unknown";

 text and unknown
------------------
 abcdef
(1 row)
```

В этом случае анализатор смотрит, есть ли оператор, у которого оба аргумента имеют тип `text`. Такой оператор находится, поэтому предполагается, что второй аргумент следует воспринимать как аргумент типа `text`.

Конкатенация двух значений неопределённых типов:

```
SELECT 'abc' || 'def' AS "unspecified";

 unspecified
-------------
 abcdef
(1 row)
```

В данном случае нет подсказки для выбора типа, так как в данном запросе никакие типы не указаны. Поэтому анализатор просматривает все возможные операторы и находит в них кандидатов, принимающих аргументы категорий string и bit-string. Так как категория string является предпочтительной, выбирается она, а затем для разрешения типа не типизированной константы выбирается предпочтительный тип этой категории, `text`.

Пример 10.3. Разрешение оператора абсолютного значения и отрицания

В каталоге операторов Postgres Pro для префиксного оператора `@` есть несколько записей, описывающих операции получения абсолютного значения для различных числовых типов данных. Одна из записей соответствует типу `float8`, предпочтительного в категории числовых типов. Таким образом, столкнувшись со значением типа `unknown`, Postgres Pro выберет эту запись:

```
SELECT @ '-4.5' AS "abs";
 abs
-----
```

```
 4.5
(1 row)
```

Здесь система неявно привела константу неизвестного типа к типу `float8`, прежде чем применять выбранный оператор. Можно убедиться в том, что выбран именно тип `float8`, а не какой-то другой:

```
SELECT @ '-4.5e500' AS "abs";

ОШИБКА:  "-4.5e500" вне диапазона для типа double precision
```

С другой стороны, префиксный оператор `~` (побитовое отрицание) определён только для целочисленных типов данных, но не для `float8`. Поэтому, если попытаться выполнить похожий запрос с `~`, мы получаем:

```
SELECT ~ '20' AS "negation";

ОШИБКА: оператор не уникален: ~ "unknown"
ПОДСКАЗКА: Не удалось выбрать лучшую кандидатуру оператора. Возможно, вам следует
добавить явные преобразования типов.
```

Это происходит оттого, что система не может решить, какой оператор предпочесть из нескольких возможных вариантов `~`. Мы можем облегчить её задачу, добавив явное преобразование:

```
SELECT ~ CAST('20' AS int8) AS "negation";

 negation
----------
      -21
(1 row)
```

Пример 10.4. Разрешение оператора включения в массив

Ещё один пример разрешения оператора с одним аргументом известного типа и другим неизвестного:

```
SELECT array[1,2] <@ '{1,2,3}' as "is subset";

 is subset
-----------
 t
(1 row)
```

В каталоге операторов Postgres Pro есть несколько записей для инфиксного оператора `<@`, но только два из них могут принять целочисленный массива слева: оператор включения массива (`anyarray<@anyarray`) и оператор включения диапазона (`anyelement<@anyrange`). Так как ни один из этих полиморфных псевдотипов (см. Раздел 8.20) не считается предпочтительным, анализатор не может избавиться от неоднозначности на данном этапе. Однако в Шаг 3.f говорится, что константа неизвестного типа должна рассматриваться как значение типа другого аргумента, в данном случае это целочисленный массив. После этого подходящим считается только один из двух операторов, так что выбирается оператор с целочисленными массивами. (Если бы был выбран оператор включения диапазона, мы получили бы ошибку, так как значение в строке не соответствует формату значений диапазона.)

Пример 10.5. Нестандартный оператор с доменом

Иногда пользователи пытаются ввести операторы, применимые только к определённому домену. Это возможно, но вовсе не так полезно, как может показаться, ведь правила разрешения операторов применяются к базовому типу домена. Взгляните на этот пример:

```
CREATE DOMAIN mytext AS text CHECK(...);
CREATE FUNCTION mytext_eq_text (mytext, text) RETURNS boolean AS ...;
CREATE OPERATOR = (procedure=mytext_eq_text, leftarg=mytext, rightarg=text);
```

```
CREATE TABLE mytable (val mytext);

SELECT * FROM mytable WHERE val = 'foo';
```

В этом запросе не будет использоваться нововведённый оператор. При разборе запроса сначала будет проверено, есть ли оператор `mytext = mytext` (см. Шаг 2.a), но это не так; затем будет рассмотрен базовый тип домена (`text`) и проверено наличие оператора `text = text` (см. Шаг 2.b), и таковой действительно есть; в итоге строковое значение типа `unknown` будет воспринято как `text` и будет применён оператор `text = text`. Единственный вариант задействовать нововведённый оператор — добавить явное приведение:

```
SELECT * FROM mytable WHERE val = text 'foo';
```

так, чтобы оператор `mytext = text` был найден сразу, согласно правилу точного совпадения. Если дело доходит до правил наибольшего соответствия, они активно дискредитируют операторы доменных типов. Если бы они этого не делали, с таким оператором возникало бы слишком много ошибок разрешения операторов, потому что правила приведения всегда считают домен приводимым к базовому типу и наоборот, так что доменный оператор применялся бы во всех случаях, где применяется одноимённый оператор с базовым типом.

10.3. Функции

При выборе конкретной функции, задействованной в выражении, PostgreSQL следует описанному ниже алгоритму.

Разрешение функции по типу

1. Выбрать функции для рассмотрения из системного каталога `pg_proc`. Если имя функции не дополнено именем схемы, будут рассматриваться все функции с подходящим именем и числом аргументов, видимые в текущем пути поиска (см. Подраздел 5.8.3). Если имя функции определено полностью, в рассмотрение принимаются только функции из указанной схемы.

   a. (Optional) Если в пути поиска оказывается несколько функций с одинаковыми типами аргументов, учитываются только те из них, которые находятся в пути раньше. Функции с разными типами аргументов рассматриваются на равных правах вне зависимости от их положения в пути поиска.

   b. (Optional) Если в числе параметров функции есть массив `VARIADIC` и при вызове не указывается ключевое слово `VARIADIC`, функция обрабатывается, как если бы этот параметр был заменён одним или несколькими параметрами типа элементов массива, по числу аргументов при вызове. После такого расширения по фактическим типам аргументов она может совпасть с некоторой функцией с постоянным числом аргументов. В этом случае используется функция, которая находится в пути раньше, а если они оказываются в одной схеме, предпочитается вариант с постоянными аргументами.

      Это создаёт угрозу безопасности при вызове с полным именем [2] функции с переменным числом аргументов, которая может оказаться в схеме, где могут создавать объекты недоверенные пользователи. Злонамеренный пользователь может перехватывать управление и выполнять произвольные SQL-функции, как будто их выполняете вы. Запись вызова с ключевым словом `VARIADIC` устраняет эту угрозу. Однако для вызовов с передачей параметров `VARIADIC "any"` часто не существует необходимой формулировки с ключом `VARIADIC`. Чтобы такие вызовы были безопасными, создание объектов в схеме функции должно разрешаться только доверенным пользователям.

   c. (Optional) Функции, для которых определены значения параметров по умолчанию, считаются совпадающими с вызовом, в котором опущено ноль или более параметров в соответствующих позициях. Если для вызова подходят несколько функций, используется та, что обнаруживается в пути поиска раньше. Если в одной схеме оказываются несколько функций с одинаковыми типами в позициях обязательных параметров (что возможно,

[2] Эта угроза неактуальна для имён без схемы, так как путь поиска, содержащий схемы, в которых недоверенные пользователи могут создавать объекты, не соответствует шаблону безопасного использования схем.

если в них определены разные наборы пропускаемых параметров), система не сможет выбрать оптимальную, и выдаст ошибку «неоднозначный вызов функции», если лучшее соответствие для вызова не будет найдено.

Это создаёт угрозу при вызове с полным именем [2] любой функции, которая может оказаться в схеме, где могут создавать объекты недоверенные пользователи. Злонамеренный пользователь может создать функцию с именем уже существующей, продублировав параметры исходной и добавив дополнительные со значениями по умолчанию. В результате при последующих вызовах будет выполняться не исходная функция. Для ликвидации этой угрозы помещайте функции в схемы, в которых создавать объекты могут только доверенные объекты.

2. Проверить, нет ли функции, принимающей в точности типы входных аргументов. Если такая функция есть (она может быть только одной в отобранном ранее наборе), использовать её. Отсутствие точного совпадения создаёт угрозу вызова с полным именем [2] функции в схеме, где могут создавать объекты недоверенные пользователи. В таких ситуациях приведите типы аргументов для получения точного соответствия. (В случаях с `unknown` совпадения на этом этапе не будет никогда.)
3. Если точное совпадение не найдено, проверить, не похож ли вызов функции на особую форму преобразования типов. Это имеет место, когда при вызове функции передаётся всего один аргумент и имя функции совпадает с именем (внутренним) некоторого типа данных. Более того, аргументом функции должна быть либо строка неопределённого типа, либо значение типа, двоично-совместимого с указанным или приводимого к нему с помощью функций ввода/вывода типа (то есть, преобразований в стандартный строковый тип и обратно). Если эти условия выполняются, вызов функции воспринимается как особая форма конструкции `CAST`. [3]
4. Найти самый подходящий.
   a. Отбросить кандидатов, для которых входные типы не совпадают и не могут быть преобразованы (неявным образом) так, чтобы они совпали. В данном случае считается, что константы типа `unknown` можно преобразовать во что угодно. Если остаётся только один кандидат, использовать его, в противном случае перейти к следующему шагу.
   b. Если один из аргументов имеет тип домен, далее считать его типом базовый тип домена. Благодаря этому при поиске неоднозначно заданной функции домены будут подобны свои базовым типам.
   c. Просмотреть всех кандидатов и оставить только тех, для которых точно совпадают как можно больше типов аргументов. Оставить всех кандидатов, если точных совпадений нет. Если остаётся только один кандидат, использовать его, в противном случае перейти к следующему шагу.
   d. Просмотреть всех кандидатов и оставить только тех, которые принимают предпочитаемые типы (из категории типов входных значений) в наибольшем числе позиций, где требуется преобразование типов. Оставить всех кандидатов, если ни один не принимает предпочитаемые типы. Если остаётся только один кандидат, использовать его, в противном случае перейти к следующему шагу.
   e. Если какие-либо значения имеют тип `unknown`, проверить категории типов, принимаемых в данных позициях аргументов оставшимися кандидатами. Для каждой позиции выбрать категорию `string`, если какой-либо кандидат принимает эту категорию. (Эта склонность к строкам объясняется тем, что константа типа unknown выглядит как строка.) Если эта категория не подходит, но все оставшиеся кандидаты принимают одну категорию, выбрать её; в противном случае констатировать неудачу — сделать правильный выбор без дополнительных подсказок нельзя. Затем отбросить кандидатов, которые не принимают типы выбранной категории. Далее, если какой-либо кандидат принимает предпочитаемый тип из этой категории, отбросить кандидатов, принимающих другие, не предпочитаемые типы для данного аргумента. Оставить всех кандидатов, если эти проверки не прошёл ни

[3] Этот шаг нужен для поддержки приведений типов в стиле вызова функции, когда на самом деле соответствующей функции приведения нет. Если такая функция приведения есть, она обычно называется именем выходного типа и необходимости в особом подходе нет. За дополнительными комментариями обратитесь к CREATE CAST.

один. Если остаётся только один кандидат, использовать его, в противном случае перейти к следующему шагу.

f. Если в списке аргументов есть аргументы и типа `unknown`, и известного типа, и этот известный тип один для всех аргументов, предположить, что аргументы типа `unknown` также имеют этот тип, и проверить, какие кандидаты могут принимать этот тип в позиции аргумента `unknown`. Если остаётся только один кандидат, использовать его, в противном случае констатировать неудачу.

Заметьте, что для функций действуют те же правила «оптимального соответствия», что и для операторов. Они проиллюстрированы следующими примерами.

Пример 10.6. Разрешение функции округления по типам аргументов

В PostgreSQL есть только одна функция `round`, принимающая два аргумента: первый типа `numeric`, а второй — `integer`. Поэтому в следующем запросе первый аргумент `integer` автоматически приводится к типу `numeric`:

```
SELECT round(4, 4);

 round
--------
 4.0000
(1 row)
```

Таким образом, анализатор преобразует этот запрос в:

```
SELECT round(CAST (4 AS numeric), 4);
```

Так как числовые константы с десятичными точками изначально относятся к типу `numeric`, для следующего запроса преобразование типов не потребуется, так что он немного эффективнее:

```
SELECT round(4.0, 4);
```

Пример 10.7. Разрешение функций с переменными параметрами

```
CREATE FUNCTION public.variadic_example(VARIADIC numeric[]) RETURNS int
  LANGUAGE sql AS 'SELECT 1';
CREATE FUNCTION
```

Эта функция принимает в аргументах ключевое слово VARIADIC, но может вызываться и без него. Ей можно передавать и целочисленные, и любые числовые аргументы:

```
SELECT public.variadic_example(0),
       public.variadic_example(0.0),
       public.variadic_example(VARIADIC array[0.0]);
 variadic_example | variadic_example | variadic_example
------------------+------------------+------------------
                1 |                1 |                1
(1 row)
```

Однако для первого и второго вызова предпочтительнее окажутся специализированные функции, если таковые есть:

```
CREATE FUNCTION public.variadic_example(numeric) RETURNS int
  LANGUAGE sql AS 'SELECT 2';
CREATE FUNCTION

CREATE FUNCTION public.variadic_example(int) RETURNS int
  LANGUAGE sql AS 'SELECT 3';
CREATE FUNCTION

SELECT public.variadic_example(0),
       public.variadic_example(0.0),
       public.variadic_example(VARIADIC array[0.0]);
```

```
 variadic_example | variadic_example | variadic_example
------------------+------------------+------------------
                3 |                2 |                1
(1 row)
```

Если используется конфигурация по умолчанию и существует только первая функция, первый и второй вызовы будут небезопасными. Любой пользователь может перехватить их, создав вторую или третью функцию. Безопасным будет третий вызов, в котором тип аргумента соответствует в точности и используется ключевое слово `VARIADIC`.

Пример 10.8. Разрешение функции извлечения подстроки

В PostgreSQL есть несколько вариантов функции `substr`, и один из них принимает аргументы типов `text` и `integer`. Если эта функция вызывается со строковой константой неопределённого типа, система выбирает функцию, принимающую аргумент предпочитаемой категории `string` (а конкретнее, типа `text`).

```
SELECT substr('1234', 3);

 substr
--------
     34
(1 row)
```

Если текстовая строка имеет тип `varchar`, например когда данные поступают из таблицы, анализатор попытается привести её к типу `text`:

```
SELECT substr (varchar '1234', 3);

 substr
--------
     34
(1 row)
```

Этот запрос анализатор фактически преобразует в:

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

> **Примечание**
>
> Анализатор узнаёт из каталога `pg_cast`, что типы `text` и `varchar` двоично-совместимы, что означает, что один тип можно передать функции, принимающей другой, не выполняя физического преобразования. Таким образом, в данном случае операция преобразования на самом не добавляется.

И если функция вызывается с аргументом типа `integer`, анализатор попытается преобразовать его в тип `text`:

```
SELECT substr(1234, 3);
ОШИБКА: функция substr(integer, integer) не существует
ПОДСКАЗКА: Функция с данными именем и типами аргументов не найдена. Возможно, вам
следует добавить явные преобразования типов.
```

Этот вариант не работает, так как `integer` нельзя неявно преобразовать в `text`. Однако с явным преобразованием запрос выполняется:

```
SELECT substr(CAST (1234 AS text), 3);

 substr
--------
     34
```

```
(1 row)
```

10.4. Хранимое значение

Значения, вставляемые в таблицу, преобразуется в тип данных целевого столбца по следующему алгоритму.

Преобразование по типу хранения

1. Проверить точное совпадение с целевым типом.
2. Если типы не совпадают, попытаться привести тип к целевому. Это возможно, если в каталоге `pg_cast` (см. CREATE CAST) зарегистрировано *приведение присваивания* между двумя типами. Если же результат выражения — строка неизвестного типа, содержимое этой строки будет подано на вход процедуре ввода целевого типа.
3. Проверить, не требуется ли приведение размера для целевого типа. Приведение размера — это преобразование типа к такому же. Если это приведение описано в каталоге `pg_cast`, применить к его к результату выражения, прежде чем сохранить в целевом столбце. Функция, реализующая такое приведение, всегда принимает дополнительный параметр типа `integer`, в котором передаётся значение `atttypmod` для целевого столбца (обычно это её объявленный размер, хотя интерпретироваться значение `atttypmod` для разных типов данных может по-разному), и третий параметр типа `boolean`, передающий признак явное/неявное преобразование. Функция приведения отвечает за все операции с длиной, включая её проверку и усечение данных.

Пример 10.9. Преобразование для типа хранения `character`

Следующие запросы показывают, что сохраняемое значение подгоняется под размер целевого столбца, объявленного как `character(20)`:

```
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, octet_length(v) FROM vv;

          v           | octet_length
----------------------+--------------
 abcdef               |           20
(1 row)
```

Суть происходящего здесь в том, что две константы неизвестного типа по умолчанию воспринимаются как значения `text`, что позволяет применить к ним оператор `||` как оператор конкатенации значений `text`. Затем результат оператора, имеющий тип `text`, приводится к типу `bpchar` («blank-padded char» (символы, дополненные пробелами), внутреннее имя типа `character`) в соответствии с типом целевого столбца. (Так как типы `text` и `bpchar` двоично-совместимы, при этом преобразовании реальный вызов функции не добавляется.) Наконец, в системном каталоге находится функция изменения размера `bpchar(bpchar, integer, boolean)` и применяется для результата оператора и длины столбца. Эта связанная с типом функция проверяет длину данных и добавляет недостающие пробелы.

10.5. `UNION`, `CASE` и связанные конструкции

SQL-конструкция `UNION` взаимодействует с системой типов, так как ей приходится объединять значения возможно различных типов в единый результирующий набор. Алгоритм разрешения типов при этом применяется независимо к каждому выходному столбцу запроса. Конструкции `INTERSECT` и `EXCEPT` сопоставляют различные типы подобно `UNION`. По такому же алгоритму сопоставляют типы выражений и определяют тип своего результата некоторые другие конструкции, включая `CASE`, `ARRAY`, `VALUES` и функции `GREATEST` и `LEAST`.

Разрешение типов для `UNION`, `CASE` и связанных конструкций

1. Если все данные одного типа и это не тип `unknown`, выбрать его.

2. Если тип данных — домен, далее считать их типом базовый тип домена. [4]
3. Если все данные типа `unknown`, выбрать для результата тип `text` (предпочитаемый для категории string). В противном случае значения `unknown` для остальных правил игнорируются.
4. Если известные типы входных данных оказываются не из одной категории, констатировать неудачу.
5. Выбрать первый известный тип данных в качестве типа-кандидата, затем рассмотреть все остальные известные типы данных, слева направо. [5] Если ранее выбранный тип может быть неявно преобразован к другому типу, но преобразовать второй в первый нельзя, выбрать второй тип в качестве нового кандидата. Затем продолжать рассмотрение последующих данных. Если на любом этапе этого процесса выбирается предпочитаемый тип, следующие данные больше не рассматриваются.
6. Привести все данные к окончательно выбранному типу. Констатировать неудачу, если неявное преобразование из типа входных данных в выбранный тип невозможно.

Ниже это проиллюстрировано на примерах.

Пример 10.10. Разрешение типов с частичным определением в Union

```
SELECT text 'a' AS "text" UNION SELECT 'b';

 text
------
 a
 b
(2 rows)
```

В данном случае константа `'b'` неизвестного типа будет преобразована в тип `text`.

Пример 10.11. Разрешение типов в простом объединении

```
SELECT 1.2 AS "numeric" UNION SELECT 1;

 numeric
---------
       1
     1.2
(2 rows)
```

Константа `1.2` имеет тип `numeric` и целочисленное значение `1` может быть неявно приведено к типу `numeric`, так что используется этот тип.

Пример 10.12. Разрешение типов в противоположном объединении

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);

 real
------
    1
  2.2
(2 rows)
```

Здесь значение типа `real` нельзя неявно привести к `integer`, но `integer` можно неявно привести к `real`, поэтому типом результата объединения будет `real`.

Пример 10.13. Разрешение типов во вложенном объединении

```
SELECT NULL UNION SELECT NULL UNION SELECT 1;
```

[4] Так же, как домены воспринимаются при выборе операторов и функций, доменные типы могут сохраняться в конструкции `UNION` или подобной, если пользователь позаботится о том, чтобы все входные данные приводились к этому типу явно или неявно. В противном случае будет использоваться базовый тип домена.

[5] По историческим причинам в конструкции `CASE` выражение в предложении `ELSE` (если оно есть) обрабатывается как «первое», а предложения `THEN` рассматриваются после. Во всех остальных случаях, «слева направо» означает порядок, в котором выражения действительно идут в тексте запроса.

```
ERROR:  UNION types text and integer cannot be matched
```

Эта ошибка возникает из-за того, что Postgres Pro воспринимает множественные UNION как пары с вложенными операциями, то есть как запись

```
(SELECT NULL UNION SELECT NULL) UNION SELECT 1;
```

Внутренний UNION разрешается как выдающий тип text, согласно правилам, приведённым выше. Затем внешний UNION получает на вход типы text и integer, что и приводит к показанной ошибке. Эту проблему можно устранить, сделав так, чтобы у самого левого UNION минимум с одной стороны были данные желаемого типа результата.

Операции INTERSECT и EXCEPT также разрешаются по парам. Однако остальные конструкции, описанные в этом разделе, рассматривают все входные данные сразу.

10.6. Выходные столбцы SELECT

Правила, описанные в предыдущих разделах, распространяются на присваивание типов данных, кроме unknown, во всех выражениях в запросах SQL, за исключением бестиповых буквальных значений, принимающих вид простых выходных столбцов команды SELECT. Например, в запросе

```
SELECT 'Hello World';
```

ничто не говорит о том, какой тип должно принимать строковое буквальное значение. В этой ситуации Postgres Pro разрешит тип такого значения как text.

Когда SELECT является одной из ветвей конструкции UNION (или INTERSECT/EXCEPT) или когда он находится внутри INSERT ... SELECT, это правило не действует, так как более высокий приоритет имеют правила, описанные в предыдущих разделах. В первом случае тип бестипового буквального значения может быть получен из другой ветви UNION, а во втором — из целевого столбца.

Списки RETURNING в данном контексте воспринимаются так же, как выходные списки SELECT.

> **Примечание**
>
> До Postgres Pro 10 этого правила не было и бестиповые буквальные значения в выходном списке SELECT оставались с типом unknown. Это имело различные негативные последствия, так что было решено это изменить.

Глава 11. Индексы

Индексы — это традиционное средство увеличения производительности БД. Используя индекс, сервер баз данных может находить и извлекать нужные строки гораздо быстрее, чем без него. Однако с индексами связана дополнительная нагрузка на СУБД в целом, поэтому применять их следует обдуманно.

11.1. Введение

Предположим, что у нас есть такая таблица:

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

и приложение выполняет много подобных запросов:

```
SELECT content FROM test1 WHERE id = константа;
```

Если система не будет заранее подготовлена, ей придётся сканировать всю таблицу `test1`, строку за строкой, чтобы найти все подходящие записи. Когда таблица `test1` содержит большое количество записей, а этот запрос должен вернуть всего несколько (возможно, одну или ноль), такое сканирование, очевидно, неэффективно. Но если создать в системе индекс по полю `id`, она сможет находить строки гораздо быстрее. Возможно, для этого ей понадобится опуститься всего на несколько уровней в дереве поиска.

Подобный подход часто используется в технической литературе: термины и понятия, которые могут представлять интерес, собираются в алфавитном указателе в конце книги. Читатель может просмотреть этот указатель довольно быстро и затем перейти сразу к соответствующей странице, вместо того, чтобы пролистывать всю книгу в поисках нужного материала. Так же, как задача автора предугадать, что именно будут искать в книге читатели, задача программиста баз данных — заранее определить, какие индексы будут полезны.

Создать индекс для столбца `id` рассмотренной ранее таблицы можно с помощью следующей команды:

```
CREATE INDEX test1_id_index ON test1 (id);
```

Имя индекса `test1_id_index` может быть произвольным, главное, чтобы оно позволяло понять, для чего этот индекс.

Для удаления индекса используется команда `DROP INDEX`. Добавлять и удалять индексы можно в любое время.

Когда индекс создан, никакие дополнительные действия не требуются: система сама будет обновлять его при изменении данных в таблице и сама будет использовать его в запросах, где, по её мнению, это будет эффективнее, чем сканирование всей таблицы. Вам, возможно, придётся только периодически запускать команду `ANALYZE` для обновления статистических данных, на основе которых планировщик запросов принимает решения. В Главе 14 вы можете узнать, как определить, используется ли определённый индекс и при каких условиях планировщик может решить *не* использовать его.

Индексы могут быть полезны также при выполнении команд `UPDATE` и `DELETE` с условиями поиска. Кроме того, они могут применяться в поиске с соединением. То есть, индекс, определённый для столбца, участвующего в условии соединения, может значительно ускорить запросы с JOIN.

Создание индекса для большой таблицы может занимать много времени. По умолчанию Postgres Pro позволяет параллельно с созданием индекса выполнять чтение (операторы `SELECT`) таблицы, но операции записи (`INSERT`, `UPDATE` и `DELETE`) блокируются до окончания построения индекса. Для производственной среды это ограничение часто бывает неприемлемым. Хотя есть возможность

разрешить запись параллельно с созданием индексов, при этом нужно учитывать ряд оговорок — они описаны в подразделе «Неблокирующее построение индексов».

После создания индекса система должна поддерживать его в состоянии, соответствующем данным таблицы. С этим связаны неизбежные накладные расходы при изменении данных. Таким образом, индексы, которые используются в запросах редко или вообще никогда, должны быть удалены.

11.2. Типы индексов

Postgres Pro поддерживает несколько типов индексов: B-дерево, хеш, GiST, SP-GiST, GIN, BRIN и расширение bloom. Для разных типов индексов применяются разные алгоритмы, ориентированные на определённые типы запросов. По умолчанию команда `CREATE INDEX` создаёт индексы типа B-дерево, эффективные в большинстве случаев.

B-деревья могут работать в условиях на равенство и в проверках диапазонов с данными, которые можно отсортировать в некотором порядке. Точнее, планировщик запросов Postgres Pro может задействовать индекс-B-дерево, когда индексируемый столбец участвует в сравнении с одним из следующих операторов:

```
<
<=
=
>=
>
```

При обработке конструкций, представимых как сочетание этих операторов, например `BETWEEN` и `IN`, так же может выполняться поиск по индексу-B-дереву. Кроме того, такие индексы могут использоваться и в условиях `IS NULL` и `IS NOT NULL` по индексированным столбцам.

Также оптимизатор может использовать эти индексы в запросах с операторами сравнения по шаблону `LIKE` и `~`, *если* этот шаблон определяется константой и он привязан к началу строки — например, `col LIKE 'foo%'` или `col ~ '^foo'`, но не `col LIKE '%bar'`. Но если ваша база данных использует не локаль C, для поддержки индексирования запросов с шаблонами вам потребуется создать индекс со специальным классом операторов; см. Раздел 11.9. Индексы-B-деревья можно использовать и для `ILIKE` и `~*`, но только если шаблон начинается не с алфавитных символов, то есть символов, не подверженных преобразованию регистра.

B-деревья могут также применяться для получения данных, отсортированных по порядку. Это не всегда быстрее простого сканирования и сортировки, но иногда бывает полезно.

Хеш-индексы работают только с простыми условиями равенства. Планировщик запросов может применить хеш-индекс, только если индексируемый столбец участвует в сравнении с оператором =. Создать такой индекс можно следующей командой:

```
CREATE INDEX имя ON таблица USING HASH (столбец);
```

GiST-индексы представляют собой не просто разновидность индексов, а инфраструктуру, позволяющую реализовать много разных стратегий индексирования. Как следствие, GiST-индексы могут применяться с разными операторами, в зависимости от стратегии индексирования (*класса операторов*). Например, стандартный дистрибутив Postgres Pro включает классы операторов GiST для нескольких двумерных типов геометрических данных, что позволяет применять индексы в запросах с операторами:

```
<<
&<
&>
>>
<<|
&<|
|&>
```

```
|>>
@>
<@
~=
&&
```

(Эти операторы описаны в Разделе 9.11.) Классы операторов GiST, включённые в стандартный дистрибутив, описаны в Таблице 58.1. В коллекции `contrib` можно найти и другие классы операторов GiST, реализованные как отдельные проекты. За дополнительными сведениями обратитесь к Главе 58.

GiST-индексы также могут оптимизировать поиск «ближайшего соседа», например такой:

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

который возвращает десять расположений, ближайших к заданной точке. Возможность такого применения индекса опять же зависит от класса используемого оператора. Операторы, которые можно использовать таким образом, перечислены в Таблице 58.1, в столбце «Операторы сортировки».

Индексы SP-GiST, как и GiST, предоставляют инфраструктуру, поддерживающую различные типы поиска. SP-GiST позволяет организовывать на диске самые разные несбалансированные структуры данных, такие как деревья квадрантов, k-мерные и префиксные деревья. Например, стандартный дистрибутив Postgres Pro включает классы операторов SP-GiST для точек в двумерном пространстве, что позволяет применять индексы в запросах с операторами:

```
<<
>>
~=
<@
<^
>^
```

(Эти операторы описаны в Разделе 9.11.) Классы операторов SP-GiST, включённые в стандартный дистрибутив, описаны в Таблице 59.1. За дополнительными сведениями обратитесь к Главе 59.

GIN-индексы представляют собой «инвертированные индексы», в которых могут содержаться значения с несколькими ключами, например массивы. Инвертированный индекс содержит отдельный элемент для значения каждого компонента, и может эффективно работать в запросах, проверяющих присутствие определённых значений компонентов.

Подобно GiST и SP-GiST, индексы GIN могут поддерживать различные определённые пользователем стратегии и в зависимости от них могут применяться с разными операторами. Например, стандартный дистрибутив Postgres Pro включает класс операторов GIN для массивов, что позволяет применять индексы в запросах с операторами:

```
<@
@>
=
&&
```

(Эти операторы описаны в Разделе 9.18.) Классы операторов GIN, включённые в стандартный дистрибутив, описаны в Таблице 60.1. В коллекции `contrib` и в отдельных проектах можно найти и много других классов операторов GIN. За дополнительными сведениями обратитесь к Главе 60.

BRIN-индексы (сокращение от Block Range INdexes, Индексы зон блоков) хранят обобщённые сведения о значениях, находящихся в физически последовательно расположенных блоках таблицы. Подобно GiST, SP-GiST и GIN, индексы BRIN могут поддерживать определённые пользователем стратегии, и в зависимости от них применяться с разными операторами. Для типов данных, имеющих линейный порядок сортировки, записям в индексе соответствуют минимальные

и максимальные значения данных в столбце для каждой зоны блоков. Это позволяет поддерживать запросы со следующими операторами:

```
<
<=
=
>=
>
```

Классы операторов BRIN, включённые в стандартный дистрибутив, описаны в Таблице 61.1. За дополнительными сведениями обратитесь к Главе 61.

11.3. Составные индексы

Индексы можно создавать и по нескольким столбцам таблицы. Например, если у вас есть таблица:

```
CREATE TABLE test2 (
  major int,
  minor int,
  name varchar
);
```

(предположим, что вы поместили в неё содержимое каталога `/dev`) и вы часто выполняете запросы вида:

```
SELECT name FROM test2 WHERE major = константа AND minor = константа;
```

тогда имеет смысл определить индекс, покрывающий оба столбца `major` и `minor`. Например:

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

В настоящее время составными могут быть только индексы типов B-дерево, GiST, GIN и BRIN. Число столбцов в индексе ограничивается 32. (Этот предел можно изменить при компиляции Postgres Pro; см. файл `pg_config_manual.h`.)

Составной индекс-B-дерево может применяться в условиях с любым подмножеством столбцов индекса, но наиболее эффективен он при ограничениях по ведущим (левым) столбцам. Точное правило состоит в том, что сканируемая область индекса определяется условиями равенства с ведущими столбцами и условиями неравенства с первым столбцом, не участвующим в условии равенства. Ограничения столбцов правее них также проверяются по индексу, так что обращение к таблице откладывается, но на размер сканируемой области индекса это уже не влияет. Например, если есть индекс по столбцам `(a, b, c)` и условие `WHERE a = 5 AND b >= 42 AND c < 77`, индекс будет сканироваться от первой записи `a` = 5 и `b` = 42 до последней с `a` = 5. Записи индекса, в которых `c` >= 77, не будут учитываться, но тем не менее будут просканированы. Этот индекс в принципе может использоваться в запросах с ограничениями по `b` и/или `c`, без ограничений столбца `a`, но при этом будет просканирован весь индекс, так что в большинстве случаев планировщик предпочтёт использованию индекса полное сканирование таблицы.

Составной индекс GiST может применяться в условиях с любым подмножеством столбцов индекса. Условия с дополнительными столбцами ограничивают записи, возвращаемые индексом, но в первую очередь сканируемая область индекса определяется ограничением первого столбца. GiST-индекс будет относительно малоэффективен, когда первый его столбец содержит только несколько различающихся значений, даже если дополнительные столбцы дают множество различных значений.

Составной индекс GIN может применяться в условиях с любым подмножеством столбцов индекса. В отличие от индексов GiST или B-деревьев, эффективность поиска по нему не меняется в зависимости от того, какие из его столбцов используются в условиях запроса.

Составной индекс BRIN может применяться в условиях запроса с любым подмножеством столбцов индекса. Подобно индексу GIN и в отличие от B-деревьев или GiST, эффективность поиска по нему не меняется в зависимости от того, какие из его столбцов используются в условиях

запроса. Единственное, зачем в одной таблице могут потребоваться несколько индексов BRIN вместо одного составного индекса — это затем, чтобы применялись разные параметры хранения `pages_per_range`.

При этом, разумеется, каждый столбец должен использоваться с операторами, соответствующими типу индекса; ограничения с другими операторами рассматриваться не будут.

Составные индексы следует использовать обдуманно. В большинстве случаев индекс по одному столбцу будет работать достаточно хорошо и сэкономит время и место. Индексы по более чем трём столбцам вряд ли будут полезными, если только таблица не используется крайне однообразно. Описание достоинств различных конфигураций индексов можно найти в Разделе 11.5 и Разделе 11.11.

11.4. Индексы и предложения `ORDER BY`

Помимо простого поиска строк для выдачи в результате запроса, индексы также могут применяться для сортировки строк в определённом порядке. Это позволяет учесть предложение `ORDER BY` в запросе, не выполняя сортировку дополнительно. Из всех типов индексов, которые поддерживает Postgres Pro, сортировать данные могут только B-деревья — индексы других типов возвращают строки в неопределённом, зависящем от реализации порядке.

Планировщик может выполнить указание `ORDER BY`, либо просканировав существующий индекс, подходящий этому указанию, либо просканировав таблицу в физическом порядке и выполнив сортировку явно. Для запроса, требующего сканирования большой части таблицы, явная сортировка скорее всего будет быстрее, чем применение индекса, так как при последовательном чтении она потребует меньше операций ввода/вывода. Важный особый случай представляет `ORDER BY` в сочетании с `LIMIT` *n*: при явной сортировке системе потребуется обработать все данные, чтобы выбрать первые *n* строк, но при наличии индекса, соответствующего столбцам в `ORDER BY`, первые *n* строк можно получить сразу, не просматривая остальные вовсе.

По умолчанию элементы B-дерева хранятся в порядке возрастания, при этом значения NULL идут в конце. Это означает, что при прямом сканировании индекса по столбцу `x` порядок оказывается соответствующим указанию `ORDER BY x` (или точнее, `ORDER BY x ASC NULLS LAST`). Индекс также может сканироваться в обратную сторону, и тогда порядок соответствует указанию `ORDER BY x DESC` (или точнее, `ORDER BY x DESC NULLS FIRST`, так как для `ORDER BY DESC` подразумевается `NULLS FIRST`).

Вы можете изменить порядок сортировки элементов B-дерева, добавив уточнения `ASC`, `DESC`, `NULLS FIRST` и/или `NULLS LAST` при создании индекса; например:

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

Индекс, в котором элементы хранятся в порядке возрастания и значения NULL идут первыми, может удовлетворять указаниям `ORDER BY x ASC NULLS FIRST` или `ORDER BY x DESC NULLS LAST`, в зависимости от направления просмотра.

У вас может возникнуть вопрос, зачем нужны все четыре варианта при создании индексов, когда и два варианта с учётом обратного просмотра покрывают все виды `ORDER BY`. Для индексов по одному столбцу это и в самом деле излишне, но для индексов по многим столбцам это может быть полезно. Рассмотрим индекс по двум столбцам `(x, y)`: он может удовлетворять указанию `ORDER BY x, y` при прямом сканировании или `ORDER BY x DESC, y DESC` при обратном. Но вполне возможно, что приложение будет часто выполнять `ORDER BY x ASC, y DESC`. В этом случае получить такую сортировку от простого индекса нельзя, но можно получить подходящий индекс, определив его как `(x ASC, y DESC)` или `(x DESC, y ASC)`.

Очевидно, что индексы с нестандартными правилами сортировки весьма специфичны, но иногда они могут кардинально ускорить определённые запросы. Стоит ли вводить такие индексы, зависит от того, как часто выполняются запросы с необычным порядком сортировки.

11.5. Объединение нескольких индексов

При простом сканировании индекса могут обрабатываться только те предложения в запросе, в которых применяются операторы его класса и объединяет их AND. Например, для индекса (a, b) условие запроса WHERE a = 5 AND b = 6 сможет использовать этот индекс, а запрос WHERE a = 5 OR b = 6 — нет.

К счастью, Postgres Pro способен соединять несколько индексов (и в том числе многократно применять один индекс) и охватывать также случаи, когда сканирования одного индекса недостаточно. Система может сформировать условия AND и OR за несколько проходов индекса. Например, запрос WHERE x = 42 OR x = 47 OR x = 53 OR x = 99 можно разбить на четыре сканирования индекса по x, по сканированию для каждой части условия. Затем результаты этих сканирований будут логически сложены (OR) вместе и дадут конечный результат. Другой пример — если у нас есть отдельные индексы по x и y, запрос WHERE x = 5 AND y = 6 можно выполнить, применив индексы для соответствующих частей запроса, а затем вычислив логическое произведение (AND) для найденных строк, которое и станет конечным результатом.

Выполняя объединение нескольких индексов, система сканирует все необходимые индексы и создаёт в памяти *битовую карту* расположения строк таблицы, которые удовлетворяют условиям каждого индекса. Затем битовые карты объединяются операциями AND и OR, как того требуют условия в запросе. Наконец система обращается к соответствующим отмеченным строкам таблицы и возвращает их данные. Строки таблицы просматриваются в физическом порядке, как они представлены в битовой карте; это означает, что порядок сортировки индексов при этом теряется и в запросах с предложением ORDER BY сортировка будет выполняться отдельно. По этой причине, а также потому, что каждое сканирование индекса занимает дополнительное время, планировщик иногда выбирает простое сканирование индекса, несмотря на то, что можно было бы подключить и дополнительные индексы.

В большинстве приложений (кроме самых простых) полезными могут оказаться различные комбинации индексов, поэтому разработчик баз данных, определяя набор индексов, должен искать компромиссное решение. Иногда оказываются хороши составные индексы, а иногда лучше создать отдельные индексы и положиться на возможности объединения индексов. Например, если типичную нагрузку составляют запросы иногда с условием только по столбцу x, иногда только по y, а иногда по обоим столбцам, вы можете ограничиться двумя отдельными индексами по x и y, рассчитывая на то, что при обработке условий с обоими столбцами эти индексы будут объединяться. С другой стороны, вы можете создать один составной индекс по (x, y). Этот индекс скорее всего будет работать эффективнее, чем объединение индексов, в запросах с двумя столбцами, но как говорилось в Разделе 11.3, он будет практически бесполезен для запросов с ограничениями только по y, так что одного этого индекса будет недостаточно. Выигрышным в этом случае может быть сочетание составного индекса с отдельным индексом по y. В запросах, где задействуется только x, может применяться составной индекс, хотя он будет больше и, следовательно, медленнее индекса по одному x. Наконец, можно создать все три индекса, но это будет оправдано, только если данные в таблице изменяются гораздо реже, чем выполняется поиск в таблице, при этом частота запросов этих трёх типов примерно одинакова. Если запросы какого-то одного типа выполняются гораздо реже других, возможно лучше будет оставить только два индекса, соответствующих наиболее частым запросам.

11.6. Уникальные индексы

Индексы также могут обеспечивать уникальность значения в столбце или уникальность сочетания значений в нескольких столбцах.

```
CREATE UNIQUE INDEX имя ON таблица (столбец [, ...])
[INCLUDE (столбец [, ...])];
```

В настоящее время уникальными могут быть только индексы-B-деревья.

Если индекс создаётся как уникальный, в таблицу нельзя будет добавить несколько строк с одинаковыми значениями ключа индекса. При этом значения NULL считаются не равными

друг другу. Составной уникальный индекс не принимает только те строки, в которых все индексируемые столбцы содержат одинаковые значения. Столбцы, включённые предложением INCLUDE, не учитываются при контроле ограничений (UNIQUE, PRIMARY KEY и т. д.).

Когда для таблицы определяется ограничение уникальности или первичный ключ, Postgres Pro автоматически создаёт уникальный индекс по всем столбцам, составляющим это ограничение или первичный ключ (индекс может быть составным). Такой индекс и является механизмом, который обеспечивает выполнение ограничения.

> **Примечание**
>
> Для уникальных столбцов не нужно вручную создавать отдельные индексы — они просто продублируют индексы, созданные автоматически.

11.7. Индексы по выражениям

Индекс можно создать не только по столбцу нижележащей таблицы, но и по функции или скалярному выражению с одним или несколькими столбцами таблицы. Это позволяет быстро находить данные в таблице по результатам вычислений.

Например, для сравнений без учёта регистра символов часто используется функция lower:

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

Этот запрос сможет использовать индекс, определённый для результата функции lower(col1) так:

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

Если мы объявим этот индекс уникальным (UNIQUE), он не даст добавить строки, в которых значения col1 различаются только регистром, как и те, в которых значения col1 действительно одинаковые. Таким образом, индексы по выражениям можно использовать ещё и для обеспечения ограничений, которые нельзя записать как простые ограничения уникальности.

Если же часто выполняются запросы вида:

```
SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';
```

тогда, возможно, стоит создать такой индекс:

```
CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

Синтаксис команды CREATE INDEX обычно требует заключать индексные выражения в скобки, как показано во втором примере. Если же выражение представляет собой просто вызов функции, как в первом примере, дополнительные скобки можно опустить.

Поддержка индексируемых выражений обходится довольно дорого, так как эти выражения должны вычисляться при добавлении каждой строки и при каждом изменении без оптимизации HOT. Однако при поиске по индексу индексируемое выражение *не* вычисляется повторно, так как его результат уже сохранён в индексе. В рассмотренных выше случаях система видит запрос как WHERE столбец_индекса = 'константа', и поэтому поиск выполняется так же быстро, как и с простым индексом. Таким образом, индексы по выражениям могут быть полезны, когда скорость извлечения данных гораздо важнее скорости добавления и изменения.

11.8. Частичные индексы

Частичный индекс — это индекс, который строится по подмножеству строк таблицы, определяемому условным выражением (оно называется *предикатом* частичного индекса). Такой индекс содержит записи только для строк, удовлетворяющих предикату. Частичные индексы довольно специфичны, но в ряде ситуаций они могут быть очень полезны.

Частичные индексы могут быть полезны, во-первых, тем, что позволяют избежать индексирования распространённых значений. Так как при поиске распространённого значения (такого, которое

содержится в значительном проценте всех строк) индекс всё равно не будет использоваться, хранить эти строки в индексе нет смысла. Исключив их из индекса, можно уменьшить его размер, а значит и ускорить запросы, использующие этот индекс. Это также может ускорить операции изменения данных в таблице, так как индекс будет обновляться не всегда. Возможное применение этой идеи проиллюстрировано в Примере 11.1.

Пример 11.1. Настройка частичного индекса, исключающего распространённые значения

Предположим, что вы храните в базе данных журнал обращений к корпоративному сайту. Большая часть обращений будет происходить из диапазона IP-адресов вашей компании, а остальные могут быть откуда угодно (например, к нему могут подключаться внешние сотрудники с динамическими IP). Если при поиске по IP вас обычно интересуют внешние подключения, IP-диапазон внутренней сети компании можно не включать в индекс.

Пусть у вас есть такая таблица:

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

Создать частичный индекс для нашего примера можно так:

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
WHERE NOT (client_ip > inet '192.168.100.0' AND
           client_ip < inet '192.168.100.255');
```

Так будет выглядеть типичный запрос, использующий этот индекс:

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

А следующий запрос не будет использовать этот индекс:

```
SELECT *
FROM access_log
WHERE client_ip = inet '192.168.100.23';
```

Заметьте, что при таком определении частичного индекса необходимо, чтобы распространённые значения были известны заранее, так что такие индексы лучше использовать, когда распределение данных не меняется. Хотя такие индексы можно пересоздавать время от времени, подстраиваясь под новое распределение, это значительно усложняет поддержку.

Во-вторых, частичные индексы могут быть полезны тем, что позволяют исключить из индекса значения, которые обычно не представляют интереса; это проиллюстрировано в Примере 11.2. При этом вы получаете те же преимущества, что и в предыдущем случае, но система не сможет извлечь «неинтересные» значения по этому индексу, даже если сканирование индекса может быть эффективным. Очевидно, настройка частичных индексов в таких случаях требует тщательного анализа и тестирования.

Пример 11.2. Настройка частичного индекса, исключающего неинтересные значения

Если у вас есть таблица, в которой хранятся и оплаченные, и неоплаченные счета, и при этом неоплаченные счета составляют только небольшую часть всей таблицы, но представляют наибольший интерес, производительность запросов можно увеличить, создав индекс только по неоплаченным счетам. Сделать это можно следующей командой:

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)
    WHERE billed is not true;
```

Этот индекс будет применяться, например в таком запросе:

```
SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;
```

Однако он также может применяться в запросах, где order_nr вообще не используется, например:

```
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```

Конечно, он будет не так эффективен, как мог бы быть частичный индекс по столбцу amount, так как системе придётся сканировать его целиком. Тем не менее, если неоплаченных счетов сравнительно мало, выиграть при поиске неоплаченного счёта можно и с таким частичным индексом.

Заметьте, что в таком запросе этот индекс не будет использоваться:

```
SELECT * FROM orders WHERE order_nr = 3501;
```

Счёт с номером 3501 может оказаться, как в числе неоплаченных, так и оплаченных.

Пример 11.2 также показывает, что индексируемый столбец не обязательно должен совпадать со столбцом, используемым в предикате. Postgres Pro поддерживает частичные индексы с произвольными предикатами — главное, чтобы в них фигурировали только столбцы индексируемой таблицы. Однако не забывайте, что предикат должен соответствовать условиям запросов, для оптимизации которых предназначается данный индекс. Точнее, частичный индекс будет применяться в запросе, только если система сможет понять, что условие WHERE данного запроса математически сводится к предикату индекса. Но учтите, что Postgres Pro не умеет доказывать математические утверждения об эквивалентности выражений, записанных в разных формах. (Составить программу для таких доказательств крайне сложно, и если даже это удастся, скорость её будет неприемлема для применения на практике.) Система может выявить только самые простые следствия с неравенствами; например, понять, что из «x < 1» следует «x < 2»; во всех остальных случаях условие предиката должно точно совпадать с условием в предложении WHERE, иначе индекс будет считаться неподходящим. Сопоставление условий происходит во время планирования запросов, а не во время выполнения. Как следствие, запросы с параметрами не будут работать с частичными индексами. Например, условие с параметром «x < ?» в подготовленном запросе никогда не будет сведено к «x < 2» при всех возможных значениях параметра.

Третье возможное применение частичных индексов вообще не связано с использованием индекса в запросах. Идея заключается в том, чтобы создать уникальный индекс по подмножеству строк таблицы, как в Примере 11.3. Это обеспечит уникальность среди строк, удовлетворяющих условию предиката, но никак не будет ограничивать остальные.

Пример 11.3. Настройка частичного уникального индекса

Предположим, что у нас есть таблица с результатами теста. Мы хотим, чтобы для каждого сочетания предмета и целевой темы была только одна запись об успешном результате, а неудачных попыток могло быть много. Вот как можно этого добиться:

```
CREATE TABLE tests (
    subject text,
    target text,
    success boolean,
    ...
);

CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
    WHERE success;
```

Это подход будет особенно эффективным, когда неудачных попыток будет намного больше, чем удачных. Также можно потребовать, чтобы в столбце допускался только один NULL, создав уникальный частичный индекс с ограничением IS NULL.

Наконец, с помощью частичных индексов можно также переопределять выбираемый системой план запроса. Возможно, что для данных с неудачным распределением система решит использовать индекс, тогда как на самом деле это неэффективно. В этом случае индекс можно настроить так, чтобы в подобных запросах он не работал. Обычно Postgres Pro принимает разумные решения относительно применения индексов (т. е. старается не использовать их для получения распространённых значений, так что частичный индекс в вышеприведённом примере помог

только уменьшить размер индекса, для отказа от использования индекса он не требовался), поэтому крайне неэффективный план может быть поводом для сообщения об ошибке.

Помните, что настраивая частичный индекс, вы тем самым заявляете, что знаете о данных гораздо больше, чем планировщик запросов. В частности, вы знаете, когда такой индекс может быть полезен. Это знание обязательно должно подкрепляться опытом и пониманием того, как работают индексы в Postgres Pro. В большинстве случаев преимущества частичных индексов по сравнению с обычными будут минимальными. Однако в ряде случаев эти индексы могут быть даже вредны, о чём говорится в Примере 11.4.

Пример 11.4. Не применяйте частичные индексы в качестве замены секционированию

У вас может возникнуть желание создать множество неперекрывающихся частичных индексов, например:

```
CREATE INDEX mytable_cat_1 ON mytable (data) WHERE category = 1;
CREATE INDEX mytable_cat_2 ON mytable (data) WHERE category = 2;
CREATE INDEX mytable_cat_3 ON mytable (data) WHERE category = 3;
...
CREATE INDEX mytable_cat_N ON mytable (data) WHERE category = N;
```

Но так делать не следует! Почти наверняка вам лучше использовать один не частичный индекс, объявленный так:

```
CREATE INDEX mytable_cat_data ON mytable (category, data);
```

(Поставьте первым столбец категорий, по причинам описанным в Разделе 11.3.) При поиске в большем индексе может потребоваться опуститься на несколько уровней ниже, чем при поиске в меньшем частичном, но это почти гарантированно будет дешевле, чем выбрать при планировании из всех частичных индексов подходящий. Сложность с выбором индекса объясняется тем, что система не знает, как взаимосвязаны частичные индексы, и ей придётся проверять каждый из них, чтобы понять, соответствует ли он текущему запросу.

Если ваша таблица настолько велика, что создавать один индекс кажется действительно плохой идей, рассмотрите возможность использования секционирования (см. Раздел 5.10). Когда применяется этот механизм, система понимает, что таблицы и индексы не перекрываются, и может выполнять запросы гораздо эффективнее.

Узнать о частичных индексах больше можно в следующих источниках: ston89b, olson93 и seshadri95.

11.9. Семейства и классы операторов

В определении индекса можно указать *класс операторов* для каждого столбца индекса.

```
CREATE INDEX имя ON таблица (столбец класс_операторов [параметры сортировки] [, ...]);
```

Класс операторов определяет, какие операторы будет использовать индекс для этого столбца. Например, индекс-B-дерево по столбцу `int4` будет использовать класс `int4_ops`; этот класс операторов включает операции со значениями типа `int4`. На практике часто достаточно принять класс операторов, назначенный для типа столбца классом по умолчанию. Однако для некоторых типов данных могут иметь смысл несколько разных вариантов индексирования и реализовать их как раз позволяют разные классы операторов. Например, комплексные числа можно сортировать как по вещественной части, так и по модулю. Получить два варианта индексов для них можно, определив два класса операторов для данного типа и выбрав соответствующий класс при создании индекса. Выбранный класс операторов задаст основной порядок сортировки данных (его можно уточнить, добавив параметры `COLLATE`, `ASC`/`DESC` и/или `NULLS FIRST`/`NULLS LAST`).

Помимо классов операторов по умолчанию есть ещё несколько встроенных:

- Классы операторов `text_pattern_ops`, `varchar_pattern_ops` и `bpchar_pattern_ops` поддерживают индексы-B-деревья для типов `text`, `varchar` и `char`, соответственно. От стандартных классов операторов они отличаются тем, что сравнивают значения по символам,

не применяя правила сортировки, определённые локалью. Благодаря этому они подходят для запросов с поиском по шаблону (с `LIKE` и регулярными выражениями POSIX), когда локаль базы данных не стандартная «C». Например, вы можете проиндексировать столбец `varchar` так:

```
CREATE INDEX test_index ON test_table (col varchar_pattern_ops);
```

Заметьте, что при этом также следует создать индекс с классом операторов по умолчанию, если вы хотите ускорить запросы с обычными сравнениями <, <=, > и >= за счёт применения индексов. Классы операторов *xxx*`_pattern_ops` не подходят для таких сравнений. (Однако для проверки равенств эти классы операторов вполне пригодны.) В подобных случаях для одного столбца можно создать несколько индексов с разными классами операторов. Если же вы используете локаль C, классы операторов *xxx*`_pattern_ops` вам не нужны, так как для поиска по шаблону в локали C будет достаточно индексов с классом операторов по умолчанию.

Следующий запрос выводит список всех существующих классов операторов:

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opc.opcintype::regtype AS indexed_type,
       opc.opcdefault AS is_default
    FROM pg_am am, pg_opclass opc
    WHERE opc.opcmethod = am.oid
    ORDER BY index_method, opclass_name;
```

Класс операторов на самом деле является всего лишь подмножеством большой структуры, называемой *семейством операторов*. В случаях, когда несколько типов данных ведут себя одинаково, часто имеет смысл определить операторы так, чтобы они могли использоваться с индексами сразу нескольких типов. Сделать это можно, сгруппировав классы операторов для этих типов в одном семействе операторов. Такие многоцелевые операторы, являясь членами семейства, не будут связаны с каким-либо одним его классом.

Расширенная версия предыдущего запроса показывает семейство операторов, к которому принадлежит каждый класс операторов:

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opf.opfname AS opfamily_name,
       opc.opcintype::regtype AS indexed_type,
       opc.opcdefault AS is_default
    FROM pg_am am, pg_opclass opc, pg_opfamily opf
    WHERE opc.opcmethod = am.oid AND
          opc.opcfamily = opf.oid
    ORDER BY index_method, opclass_name;
```

Этот запрос выводит все существующие семейства операторов и все операторы, включённые в эти семейства:

```
SELECT am.amname AS index_method,
       opf.opfname AS opfamily_name,
       amop.amopopr::regoperator AS opfamily_operator
    FROM pg_am am, pg_opfamily opf, pg_amop amop
    WHERE opf.opfmethod = am.oid AND
          amop.amopfamily = opf.oid
    ORDER BY index_method, opfamily_name, opfamily_operator;
```

11.10. Индексы и правила сортировки

Один индекс может поддерживать только одно правило сортировки для индексируемого столбца. Поэтому при необходимости применять разные правила сортировки могут потребоваться несколько индексов.

Рассмотрим следующие операторы:

```
CREATE TABLE test1c (
    id integer,
    content varchar COLLATE "x"
);

CREATE INDEX test1c_content_index ON test1c (content);
```

Этот индекс автоматически использует правило сортировки нижележащего столбца. И запрос вида

```
SELECT * FROM test1c WHERE content > константа;
```

сможет использовать этот индекс, так как при сравнении по умолчанию будет действовать правило сортировки столбца. Однако этот индекс не поможет ускорить запросы с каким-либо другим правилом сортировки. Поэтому, если интерес представляют также и запросы вроде

```
SELECT * FROM test1c WHERE content > константа COLLATE "y";
```

для них можно создать дополнительный индекс, поддерживающий правило сортировки `"y"`, примерно так:

```
CREATE INDEX test1c_content_y_index ON test1c (content COLLATE "y");
```

11.11. Сканирование только индекса

Все индексы в Postgres Pro являются *вторичными*, что значит, что каждый индекс хранится вне области основных данных таблицы (которая в терминологии Postgres Pro называется *кучей* таблицы). Это значит, что при обычном сканировании индекса для извлечения каждой строки необходимо прочитать данные и из индекса, и из кучи. Более того, тогда как элементы индекса, соответствующие заданному условию `WHERE`, обычно находятся в индексе рядом, строки таблицы могут располагаться в куче произвольным образом. Таким образом, обращение к куче при поиске по индексу влечёт множество операций произвольного чтения кучи, которые могут обойтись недёшево, особенно на традиционных вращающихся носителях. (Как описано в Разделе 11.5, сканирование по битовой карте пытается снизить стоимость этих операций, упорядочивая доступ к куче, но не более того.)

Чтобы решить эту проблему с производительностью, Postgres Pro поддерживает *сканирование только индекса*, при котором результат запроса может быть получен из самого индекса, без обращения к куче. Основная идея такого сканирования в том, чтобы выдавать значения непосредственно из элемента индекса, и не обращаться к соответствующей записи в куче. Для применения этого метода есть два фундаментальных ограничения:

1. Тип индекса должен поддерживать сканирование только индекса. Индексы-B-деревья поддерживают его всегда. Индексы GiST и SP-GiST могут поддерживать его с одними классами операторов и не поддерживать с другими. Другие индексы такое сканирование не поддерживают. Суть нижележащего требования в том, что индекс должен физически хранить или каким-то образом восстанавливать исходное значение данных для каждого элемента индекса. В качестве контрпримера, индексы GIN неспособны поддерживать сканирование только индекса, так как в элементах индекса обычно хранится только часть исходного значения данных.
2. Запрос должен обращаться только к столбцам, сохранённым в индексе. Например, если в таблице построен индекс по столбцам `x` и `y`, и в ней есть также столбец `z`, такие запросы будут использовать сканирование только индекса:

   ```
   SELECT x, y FROM tab WHERE x = 'key';
   SELECT x FROM tab WHERE x = 'key' AND y < 42;
   ```

   А эти запросы не будут:

   ```
   SELECT x, z FROM tab WHERE x = 'key';
   SELECT x FROM tab WHERE x = 'key' AND z < 42;
   ```

(Индексы по выражениям и частичные индексы усложняют это правило, как описано ниже.)

Если два этих фундаментальных ограничения выполняются, то все данные, требуемые для выполнения запроса, содержатся в индексе, так что сканирование только по индексу физически возможно. Но в Postgres Pro существует и ещё одно требование для сканирования таблицы: необходимо убедиться, что все возвращаемые строки «видны» в снимке MVCC запроса, как описано в Главе 13. Информация о видимости хранится не в элементах индекса, а только в куче; поэтому на первый взгляд может показаться, что для получения данных каждой строки всё равно необходимо обращаться к куче. И это в самом деле так, если в таблице недавно произошли изменения. Однако для редко меняющихся данных есть возможность обойти эту проблему. Postgres Pro отслеживает для каждой страницы в куче таблицы, являются ли все строки в этой странице достаточно старыми, чтобы их видели все текущие и будущие транзакции. Это отражается в битах в *карте видимости* таблицы. Процедура сканирования только индекса, найдя потенциально подходящую запись в индексе, проверяет бит в карте видимости для соответствующей страницы в куче. Если он установлен, значит эта строка видна, и данные могут быть возвращены сразу. В противном случае придётся посетить запись строки в куче и проверить, видима ли она, так что никакого выигрыша по сравнению с обычным сканированием индекса не будет. И даже в благоприятном случае обращение к кучи не исключается совсем, а заменяется обращением к карте видимости; но так как карта видимости на четыре порядка меньше соответствующей ей области кучи, для работы с ней требуется много меньше операций физического ввода/вывода. В большинстве ситуаций карта видимости просто всё время находится в памяти.

Таким образом, тогда как сканирование только по индексу возможно лишь при выполнении двух фундаментальных требований, оно даст выигрыш, только если для значительной части страниц в куче таблицы установлены биты полной видимости. Но таблицы, в которых меняется лишь небольшая часть строк, встречаются достаточно часто, чтобы этот тип сканирования был весьма полезен на практике.

Чтобы эффективно применять возможность сканирования только индекса, можно создать индексы, в которых только первые столбцы будут соответствовать предложениям `WHERE`, а остальные столбцы будут содержать полезные данные, возвращаемые запросом. Например, если часто выполняется запрос вида:

```
SELECT y FROM tab WHERE x = 'key';
```

при традиционном подходе к ускорению таких запросов можно было бы создать индекс только по `x`. Однако индекс по `(x, y)` дал бы возможность выполнения этого запроса со сканированием только индекса. Как говорилось ранее, такой индекс был бы объёмнее и дороже в обслуживании, чем индекс только по `x`, так что этот вариант предпочтителен, только для таблиц в основном статических. Заметьте, что в объявлении индекса важно указать столбцы `(x, y)`, а не `(y, x)`, так как для большинства типов индексов (а именно, B-деревьев) поиск, при котором не ограничиваются значения ведущих столбцов индекса, не будет эффективным.

В принципе сканирование только индекса может применяться и с индексами по выражениям. Например, при наличии индекса по `f(x)`, где `x` — столбец таблицы, должно быть возможно выполнить

```
SELECT f(x) FROM tab WHERE f(x) < 1;
```

как сканирование только индекса; и это очень заманчиво, если `f()` — сложная для вычисления функция. Однако планировщик Postgres Pro в настоящее время может вести себя не очень разумно. Он считает, что запрос может выполняться со сканированием только индекса, лишь когда из индекса могут быть получены все *столбцы*, требующиеся для запроса. В этом примере `x` фигурирует только в контексте `f(x)`, но планировщик не замечает этого и решает, что сканирование только по индексу невозможно. Если сканирование только по индексу заслуживает того, эту проблему можно обойти, объявив индекс по `(f(x), x)`, где второй столбец может не использоваться на практике, но нужен для того, чтобы убедить планировщик, что сканирование только по индексу возможно. Если это делается ради предотвращения многократных вычислений `f(x)`, следует также учесть, что планировщик не обязательно свяжет с использованием индекса

упоминания `f(x)`, фигурирующие вне индексируемых предложений `WHERE`, со столбцом индекса. Обычно он это делает правильно в простых запросах, вроде показанного выше, но не в запросах с соединениями. Эти недостатки могут быть устранены в будущих версиях Postgres Pro.

С использованием частичных индексов при сканировании только по индексу тоже связаны интересные особенности. Предположим, что у нас есть частичный индекс, показанный в Примере 11.3:

```
CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
    WHERE success;
```

В принципе с ним мы можем произвести сканирование только по индексу при выполнении запроса

```
SELECT target FROM tests WHERE subject = 'some-subject' AND success;
```

Но есть одна проблема: предложение `WHERE` обращается к столбцу `success`, который отсутствует в результирующих столбцах индекса. Тем не менее сканирование только индекса возможно, так как плану не нужно перепроверять эту часть предложения `WHERE` во время выполнения: у всех записей, найденных в индексе, значение `success = true`, так что в плане его не нужно проверять явно. Postgres Pro версий 9.6 и новее распознает такую ситуацию и сможет произвести сканирование только по индексу, но старые версии неспособны на это.

11.12. Контроль использования индексов

Хотя индексы в Postgres Pro не требуют какого-либо обслуживания или настройки, это не избавляет от необходимости проверять, как и какие индексы используются на самом деле в реальных условиях. Узнать, как отдельный запрос использует индексы, можно с помощью команды EXPLAIN; её применение для этих целей описывается в Разделе 14.1. Также возможно собрать общую статистику об использовании индексов на работающем сервере, как описано в Разделе 27.2.

Вывести универсальную формулу, определяющую, какие индексы нужно создавать, довольно сложно, если вообще возможно. В предыдущих разделах рассматривались некоторые типовые ситуации, иллюстрирующие подходы к этому вопросу. Часто найти ответ на него помогают эксперименты. Ниже приведены ещё несколько советов:

- Всегда начинайте исследование с ANALYZE. Эта команда собирает статистические данные о распределении значений в таблице, которые необходимы для оценивания числа строк, возвращаемых запросов. А это число, в свою очередь, нужно планировщику, чтобы оценить реальные затраты для всевозможных планов выполнения запроса. Не имея реальной статистики, планировщик будет вынужден принять некоторые значения по умолчанию, которые почти наверняка не будут соответствовать действительности. Поэтому понять, как индекс используется приложением без предварительного запуска `ANALYZE`, практически невозможно. Подробнее это рассматривается в Подразделе 23.1.3 и Подразделе 23.1.6.
- Используйте в экспериментах реальные данные. Анализируя работу системы с тестовыми данными, вы поймёте, какие индексы нужны для тестовых данных, но не более того.

  Особенно сильно искажают картину очень маленькие наборы тестовых данных. Тогда как для извлечения 1000 строк из 100000 может быть применён индекс, для выбора 1 из 100 он вряд ли потребуется, так как 100 строк скорее всего уместятся в одну страницу данных на диске и никакой другой план не будет лучше обычного сканирования 1 страницы.

  Тем не менее, пока приложение не эксплуатируется, создавать какие-то тестовые данные всё равно нужно, и это нужно делать обдуманно. Если вы наполняете базу данных очень близкими, или наоборот, случайными значениями, либо добавляете строки в отсортированном порядке, вы получите совсем не ту статистику распределения, что дадут реальные данные.
- Когда индексы не используются, ради тестирования может быть полезно подключить их принудительно. Для этого можно воспользоваться параметрами выполнения, позволяющими выключать различные типы планов (см. Подраздел 18.7.1). Например, выключив наиболее простые планы: последовательное сканирование (`enable_seqscan`) и соединения с вложенными циклами (`enable_nestloop`), вы сможете заставить систему выбрать другой

план. Если же система продолжает выполнять сканирование или соединение с вложенными циклами, вероятно, у неё есть более серьёзная причина не использовать индекс; например, индекс может не соответствовать условию запроса. (Какие индексы работают в запросах разных типов, обсуждалось в предыдущих разделах.)

- Если система начинает использовать индекс только под принуждением, тому может быть две причины: либо система права и применять индекс в самом деле неэффективно, либо оценка стоимости применения индекса не соответствует действительности. В этом случае вам следует замерить время выполнения запроса с индексами и без них. В анализе этой ситуации может быть полезна команда `EXPLAIN ANALYZE`.
- Если выясняется, что оценка стоимости неверна, это может иметь тоже два объяснения. Общая стоимость вычисляется как произведение цены каждого узла плана для одной строки и оценки избирательности узла плана. Цены узлов при необходимости можно изменить параметрами выполнения (описанными в Подразделе 18.7.2). С другой стороны, оценка избирательности может быть неточной из-за некачественной статистики. Улучшить её можно, настроив параметры сбора статистики (см. ALTER TABLE).

  Если ваши попытки скорректировать стоимость планов не увенчаются успехом, возможно вам останется только явно заставить систему использовать нужный индекс. Вероятно, имеет смысл также связаться с разработчиками Postgres Pro, чтобы прояснить ситуацию.

Глава 12. Полнотекстовый поиск

12.1. Введение

Полнотекстовый поиск (или просто *поиск текста*) — это возможность находить *документы* на естественном языке, соответствующие *запросу*, и, возможно, дополнительно сортировать их по релевантности для этого запроса. Наиболее распространённая задача — найти все документы, содержащие *слова запроса*, и выдать их отсортированными по степени *соответствия* запросу. Понятия `запроса` и `соответствия` довольно расплывчаты и зависят от конкретного приложения. В самом простом случае `запросом` считается набор слов, а `соответствие` определяется частотой слов в документе.

Операторы текстового поиска существуют в СУБД уже многие годы. В Postgres Pro для текстовых типов данных есть операторы `~`, `~*`, `LIKE` и `ILIKE`, но им не хватает очень важных вещей, которые требуются сегодня от информационных систем:

- Нет поддержки лингвистического функционала, даже для английского языка. Возможности регулярных выражений ограничены — они не рассчитаны на работу со словоформами, например, `подходят` и `подходить`. С ними вы можете пропустить документы, которые содержат `подходят`, но, вероятно, и они представляют интерес при поиске по ключевому слову `подходить`. Конечно, можно попытаться перечислить в регулярном выражении все варианты слова, но это будет очень трудоёмко и чревато ошибками (некоторые слова могут иметь десятки словоформ).
- Они не позволяют упорядочивать результаты поиска (по релевантности), а без этого поиск неэффективен, когда находятся сотни подходящих документов.
- Они обычно выполняются медленно из-за отсутствия индексов, так как при каждом поиске приходится просматривать все документы.

Полнотекстовая индексация заключается в *предварительной обработке* документов и сохранении индекса для последующего быстрого поиска. Предварительная обработка включает следующие операции:

Разбор документов на фрагменты. При этом полезно выделить различные классы фрагментов, например, числа, слова, словосочетания, почтовые адреса и т. д., которые будут обрабатываться по-разному. В принципе классы фрагментов могут зависеть от приложения, но для большинства применений вполне подойдёт предопределённый набор классов. Эту операцию в Postgres Pro выполняет *анализатор* (parser). Вы можете использовать как стандартный анализатор, так и создавать свои, узкоспециализированные.

Преобразование фрагментов в лексемы. Лексема — это *нормализованный* фрагмент, в котором разные словоформы приведены к одной. Например, при нормализации буквы верхнего регистра приводятся к нижнему, а из слов обычно убираются окончания (в частности, `s` или `es` в английском). Благодаря этому можно находить разные формы одного слова, не вводя вручную все возможные варианты. Кроме того, на данном шаге обычно исключаются *стоп-слова*, то есть слова, настолько распространённые, что искать их нет смысла. (Другими словами, фрагменты представляют собой просто подстроки текста документа, а лексемы — это слова, имеющие ценность для индексации и поиска.) Для выполнения этого шага в Postgres Pro используются *словари*. Набор существующих стандартных словарей при необходимости можно расширять, создавая свои собственные.

Хранение документов в форме, подготовленной для поиска. Например, каждый документ может быть представлен в виде сортированного массива нормализованных лексем. Помимо лексем часто желательно хранить информацию об их положении для *ранжирования по близости*, чтобы документ, в котором слова запроса расположены «плотнее», получал более высокий ранг, чем документ с разбросанными словами.

Словари позволяют управлять нормализацией фрагментов с большой гибкостью. Создавая словари, можно:

- Определять стоп-слова, которые не будут индексироваться.

- Сопоставлять синонимы с одним словом, используя Ispell.
- Сопоставлять словосочетания с одним словом, используя тезаурус.
- Сопоставлять различные склонения слова с канонической формой, используя словарь Ispell.
- Сопоставлять различные склонения слова с канонической формой, используя стеммер Snowball.

Для хранения подготовленных документов в PostgreSQL предназначен тип данных `tsvector`, а для представления обработанных запросов — тип `tsquery` (Раздел 8.11). С этими типами данных работают целый ряд функций и операторов (Раздел 9.13), и наиболее важный из них — оператор соответствия `@@`, с которым мы познакомимся в Подразделе 12.1.2. Для ускорения полнотекстового поиска могут применяться индексы (Раздел 12.9).

12.1.1. Что такое документ?

Документ — это единица обработки в системе полнотекстового поиска; например, журнальная статья или почтовое сообщение. Система поиска текста должна уметь разбирать документы и сохранять связи лексем (ключевых слов) с содержащим их документом. Впоследствии эти связи могут использоваться для поиска документов с заданными ключевыми словами.

В контексте поиска в Postgres Pro документ — это обычно содержимое текстового поля в строке таблицы или, возможно, сочетание (объединение) таких полей, которые могут храниться в разных таблицах или формироваться динамически. Другими словами, документ для индексации может создаваться из нескольких частей и не храниться где-либо как единое целое. Например:

```
SELECT title || ' ' ||  author || ' ' ||  abstract || ' ' || body AS document
FROM messages
WHERE mid = 12;

SELECT m.title || ' ' || m.author || ' ' || m.abstract || ' ' || d.body AS document
FROM messages m, docs d
WHERE m.mid = d.did AND m.mid = 12;
```

> **Примечание**
>
> На самом деле в этих примерах запросов следует использовать функцию `coalesce`, чтобы значение `NULL` в каком-либо одном атрибуте не привело к тому, что результирующим документом окажется `NULL`.

Документы также можно хранить в обычных текстовых файлах в файловой системе. В этом случае база данных может быть просто хранилищем полнотекстового индекса и исполнителем запросов, а найденные документы будут загружаться из файловой системы по некоторым уникальным идентификаторам. Однако для загрузки внешних файлов требуются права суперпользователя или поддержка специальных функций, так что это обычно менее удобно, чем хранить все данные внутри БД Postgres Pro. Кроме того, когда всё хранится в базе данных, это упрощает доступ к метаданным документов при индексации и выводе результатов.

Для нужд текстового поиска каждый документ должен быть сведён к специальному формату `tsvector`. Поиск и ранжирование выполняется исключительно с этим представлением документа — исходный текст потребуется извлечь, только когда документ будет отобран для вывода пользователю. Поэтому мы часто подразумеваем под `tsvector` документ, тогда как этот тип, конечно, содержит только компактное представление всего документа.

12.1.2. Простое соответствие текста

Полнотекстовый поиск в Postgres Pro реализован на базе оператора соответствия `@@`, который возвращает `true`, если `tsvector` (документ) соответствует `tsquery` (запросу). Для этого оператора не важно, какой тип записан первым:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@
  'cat & rat'::tsquery;
 ?column?
----------
 t

SELECT 'fat & cow'::tsquery @@
  'a fat cat sat on a mat and ate a fat rat'::tsvector;
 ?column?
----------
 f
```

Как можно догадаться из этого примера, `tsquery` — это не просто текст, как и `tsvector`. Значение типа `tsquery` содержит искомые слова, это должны быть уже нормализованные лексемы, возможно объединённые в выражение операторами И, ИЛИ, НЕ и ПРЕДШЕСТВУЕТ. (Подробнее синтаксис описан в Подразделе 8.11.2.) Вы можете воспользоваться функциями `to_tsquery`, `plainto_tsquery` и `phraseto_tsquery`, которые могут преобразовать заданный пользователем текст в значение `tsquery`, прежде всего нормализуя слова в этом тексте. Функция `to_tsvector` подобным образом может разобрать и нормализовать текстовое содержимое документа. Так что запрос с поиском соответствия на практике выглядит скорее так:

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat');
 ?column?
----------
 t
```

Заметьте, что соответствие не будет обнаружено, если запрос записан как

```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat & rat');
 ?column?
----------
 f
```

так как слово `rats` не будет нормализовано. Элементами `tsvector` являются лексемы, предположительно уже нормализованные, так что `rats` считается не соответствующим `rat`.

Оператор `@@` также может принимать типы `text`, позволяя опустить явные преобразования текстовых строк в типы `tsvector` и `tsquery` в простых случаях. Всего есть четыре варианта этого оператора:

```
tsvector @@ tsquery
tsquery  @@ tsvector
text @@ tsquery
text @@ text
```

Первые два мы уже видели раньше. Форма `text@@tsquery` равнозначна выражению `to_tsvector(x) @@ y`, а форма `text@@text` — выражению `to_tsvector(x) @@ plainto_tsquery(y)`.

В значении `tsquery` оператор `&` (И) указывает, что оба его операнда должны присутствовать в документе, чтобы он удовлетворял запросу. Подобным образом, оператор `|` (ИЛИ) указывает, что в документе должен присутствовать минимум один из его операндов, тогда как оператор `!` (НЕ) указывает, что его операнд *не* должен присутствовать, чтобы условие удовлетворялось. Например, запросу `fat & ! rat` соответствуют документы, содержащие `fat` и не содержащие `rat`.

Фразовый поиск возможен с использованием оператора `<->` (ПРЕДШЕСТВУЕТ) типа `tsquery`, который находит соответствие, только если его операнды расположены рядом и в заданном порядке. Например:

```
SELECT to_tsvector('fatal error') @@ to_tsquery('fatal <-> error');
 ?column?
```

```
----------
 t

SELECT to_tsvector('error is not fatal') @@ to_tsquery('fatal <-> error');
 ?column?
----------
 f
```

Более общая версия оператора ПРЕДШЕСТВУЕТ имеет вид `<N>`, где *N* — целое число, выражающее разность между позициями найденных лексем. Запись `<1>` равнозначна `<->`, тогда как `<2>` допускает существование ровно одной лексемы между этими лексемами и т. д. Функция `phraseto_tsquery` задействует этот оператор для конструирования `tsquery`, который может содержать многословную фразу, включающую в себя стоп-слова. Например:

```
SELECT phraseto_tsquery('cats ate rats');
       phraseto_tsquery
-------------------------------
 'cat' <-> 'ate' <-> 'rat'

SELECT phraseto_tsquery('the cats ate the rats');
       phraseto_tsquery
-------------------------------
 'cat' <-> 'ate' <2> 'rat'
```

Особый случай, который иногда бывает полезен, представляет собой запись `<0>`, требующая, чтобы обоим лексемам соответствовало одно слово.

Сочетанием операторов `tsquery` можно управлять, применяя скобки. Без скобок операторы имеют следующие приоритеты, в порядке возрастания: `|`, `&`, `<->` и самый приоритетный — `!`.

Стоит отметить, что операторы И/ИЛИ/НЕ имеют несколько другое значение, когда они применяются в аргументах оператора ПРЕДШЕСТВУЕТ, так как в этом случае имеет значение точная позиция совпадения. Например, обычному `!x` соответствуют только документы, не содержащие `x` нигде. Но условию `!x <-> y` соответствует `y`, если оно не следует непосредственно за `x`; при вхождении `x` в любом другом месте документа он не будет исключаться из рассмотрения. Другой пример: для условия `x & y` обычно требуется, чтобы и `x`, и `y` встречались в каком-то месте документа, но для выполнения условия `(x & y) <-> z` требуется, чтобы `x` и `y` располагались в одном месте, непосредственно перед `z`. Таким образом, этот запрос отличается от `x <-> z & y <-> z`, которому удовлетворяют документы, содержащие две отдельные последовательности `x z` и `y z`. (Этот конкретный запрос в таком виде, как он записан, не имеет смысла, так как `x` и `y` не могут находиться в одном месте; но в более сложных ситуациях, например, с шаблонами поиска по маске, запросы этого вида могут быть полезны.)

12.1.3. Конфигурации

До этого мы рассматривали очень простые примеры поиска текста. Как было упомянуто выше, весь функционал текстового поиска позволяет делать гораздо больше: пропускать определённые слова (стоп-слова), обрабатывать синонимы и выполнять сложный анализ слов, например, выделять фрагменты не только по пробелам. Все эти функции управляются *конфигурациями текстового поиска*. В Postgres Pro есть набор предопределённых конфигураций для многих языков, но вы также можете создавать собственные конфигурации. (Все доступные конфигурации можно просмотреть с помощью команды `\dF` в psql.)

Подходящая конфигурация для данной среды выбирается во время установки и записывается в параметре default_text_search_config в `postgresql.conf`. Если вы используете для всего кластера одну конфигурацию текстового поиска, вам будет достаточно этого параметра в `postgresql.conf`. Если же требуется использовать в кластере разные конфигурации, но для каждой базы данных одну определённую, её можно задать командой `ALTER DATABASE ... SET`. В противном случае конфигурацию можно выбрать в рамках сеанса, определив параметр `default_text_search_config`.

У каждой функции текстового поиска, зависящей от конфигурации, есть необязательный аргумент `regconfig`, в котором можно явно указать конфигурацию для данной функции. Значение `default_text_search_config` используется, только когда этот аргумент опущен.

Для упрощения создания конфигураций текстового поиска они строятся из более простых объектов. В Postgres Pro есть четыре типа таких объектов:

- *Анализаторы текстового поиска* разделяют документ на фрагменты и классифицируют их (например, как слова или числа).
- *Словари текстового поиска* приводят фрагменты к нормализованной форме и отбрасывают стоп-слова.
- *Шаблоны текстового поиска* предоставляют функции, образующие реализацию словарей. (При создании словаря просто задаётся шаблон и набор параметров для него.)
- *Конфигурации текстового поиска* выбирают анализатор и набор словарей, который будет использоваться для нормализации фрагментов, выданных анализатором.

Анализаторы и шаблоны текстового поиска строятся из низкоуровневых функций на языке C; чтобы создать их, нужно программировать на C, а подключить их к базе данных может только суперпользователь. (В подкаталоге `contrib/` инсталляции Postgres Pro можно найти примеры дополнительных анализаторов и шаблонов.) Так как словари и конфигурации представляют собой просто наборы параметров, связывающие анализаторы и шаблоны, их можно создавать, не имея административных прав. Далее в этой главе будут приведены примеры их создания.

12.2. Таблицы и индексы

В предыдущем разделе приводились примеры, которые показывали, как можно выполнить сопоставление с простыми текстовыми константами. В этом разделе показывается, как находить текст в таблице, возможно с применением индексов.

12.2.1. Поиск в таблице

Полнотекстовый поиск можно выполнить, не применяя индекс. Следующий простой запрос выводит заголовок (`title`) каждой строки, содержащей слово `friend` в поле `body`:

```
SELECT title
FROM pgweb
WHERE to_tsvector('english', body) @@ to_tsquery('english', 'friend');
```

При этом также будут найдены связанные слова, такие как `friends` и `friendly`, так как все они сводятся к одной нормализованной лексеме.

В показанном выше примере для разбора и нормализации строки явно выбирается конфигурация `english`. Хотя параметры, задающие конфигурацию, можно опустить:

```
SELECT title
FROM pgweb
WHERE to_tsvector(body) @@ to_tsquery('friend');
```

Такой запрос будет использовать конфигурацию, заданную в параметре default_text_search_config.

В следующем более сложном примере выбираются десять документов, изменённых последними, со словами `create` и `table` в полях `title` или `body`:

```
SELECT title
FROM pgweb
WHERE to_tsvector(title || ' ' || body) @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

Чтобы найти строки, содержащие `NULL` в одном из полей, нужно воспользоваться функцией `coalesce`, но здесь мы опустили её вызовы для краткости.

Хотя такие запросы будут работать и без индекса, для большинства приложений скорость будет неприемлемой; этот подход рекомендуется только для нерегулярного поиска и динамического содержимого. Для практического применения полнотекстового поиска обычно создаются индексы.

12.2.2. Создание индексов

Для ускорения текстового поиска мы можем создать индекс GIN (см. Раздел 12.9):

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector('english', body));
```

Заметьте, что здесь используется функция `to_tsvector` с двумя аргументами. В выражениях, определяющих индексы, можно использовать только функции, в которых явно задаётся имя конфигурации текстового поиска (см. Раздел 11.7). Это объясняется тем, что содержимое индекса не должно зависеть от значения параметра default_text_search_config. В противном случае содержимое индекса может быть неактуальным, если разные его элементы `tsvector` будут создаваться с разными конфигурациями текстового поиска и нельзя будет понять, какую именно использовать. Выгрузить и восстановить такой индекс будет невозможно.

Так как при создании индекса использовалась версия `to_tsvector` с двумя аргументами, этот индекс будет использоваться только в запросах, где `to_tsvector` вызывается с двумя аргументами и во втором передаётся имя той же конфигурации. То есть, `WHERE to_tsvector('english', body) @@ 'a & b'` сможет использовать этот индекс, а `WHERE to_tsvector(body) @@ 'a & b'` — нет. Это гарантирует, что индекс будет использоваться только с той конфигурацией, с которой создавались его элементы.

Индекс можно создать более сложным образом, определив для него имя конфигурации в другом столбце таблицы, например:

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector(config_name, body));
```

где `config_name` — имя столбца в таблице `pgweb`. Так можно сохранить имя конфигурации, связанной с элементом индекса, и, таким образом, иметь в одном индексе элементы с разными конфигурациями. Это может быть полезно, например, когда в коллекции документов хранятся документы на разных языках. И в этом случае в запросах должен использоваться тот же индекс (с таким же образом задаваемой конфигурацией), например, так: `WHERE to_tsvector(config_name, body) @@ 'a & b'`.

Индексы могут создаваться даже по объединению столбцов:

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector('english', title || ' ' ||
 body));
```

Ещё один вариант — создать отдельный столбец `tsvector`, в котором сохранить результат `to_tsvector`. Следующий пример показывает, как можно подготовить для индексации объединённое содержимое столбцов `title` и `body`, применив функцию `coalesce` для получения желаемого результата, даже когда один из столбцов `NULL`:

```
ALTER TABLE pgweb ADD COLUMN textsearchable_index_col tsvector;
UPDATE pgweb SET textsearchable_index_col =
     to_tsvector('english', coalesce(title,'') || ' ' || coalesce(body,''));
```

Затем мы создаём индекс GIN для ускорения поиска:

```
CREATE INDEX textsearch_idx ON pgweb USING GIN (textsearchable_index_col);
```

Теперь мы можем быстро выполнять полнотекстовый поиск:

```
SELECT title
FROM pgweb
WHERE textsearchable_index_col @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
```

```
LIMIT 10;
```

Когда представление `tsvector` хранится в отдельном столбце, необходимо создать триггер, который будет поддерживать столбец с `tsvector` в актуальном состоянии при любых изменениях `title` или `body`. Как это сделать, рассказывается в Подразделе 12.4.3.

Хранение вычисленного выражения индекса в отдельном столбце даёт ряд преимуществ. Во-первых, для использования индекса в запросах не нужно явно указывать имя конфигурации текстового поиска. Как показано в вышеприведённом примере, в этом случае запрос может зависеть от `default_text_search_config`. Во-вторых, поиск выполняется быстрее, так как для проверки соответствия данных индексу не нужно повторно выполнять `to_tsvector`. (Это актуально больше для индексов GiST, чем для GIN; см. Раздел 12.9.) С другой стороны, схему с индексом по выражению проще реализовать и она позволяет сэкономить место на диске, так как представление `tsvector` не хранится явно.

12.3. Управление текстовым поиском

Для реализации полнотекстового поиска необходимы функции, позволяющие создать `tsvector` из документа и `tsquery` из запроса пользователя. Кроме того, результаты нужно выдавать в удобном порядке, так что нам потребуется функция, оценивающая релевантность документа для данного запроса. Важно также иметь возможность выводить найденный текст подходящим образом. В Postgres Pro есть все необходимые для этого функции.

12.3.1. Разбор документов

Для преобразования документа в тип `tsvector` Postgres Pro предоставляет функцию `to_tsvector`.

```
to_tsvector([конфигурация regconfig,] документ text) returns tsvector
```

`to_tsvector` разбирает текстовый документ на фрагменты, сводит фрагменты к лексемам и возвращает значение `tsvector`, в котором перечисляются лексемы и их позиции в документе. При обработке документа используется указанная конфигурация текстового поиска или конфигурация по умолчанию. Простой пример:

```
SELECT to_tsvector('english', 'a fat  cat sat on a mat - it ate a fat rats');
                  to_tsvector
-----------------------------------------------------
 'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

В этом примере мы видим, что результирующий `tsvector` не содержит слова `a`, `on` и `it`, слово `rats` превратилось `rat`, а знак препинания «-» был проигнорирован.

Функция `to_tsvector` внутри вызывает анализатор, который разбивает текст документа на фрагменты и классифицирует их. Для каждого фрагмента она проверяет список словарей (Раздел 12.6), определяемый типом фрагмента. Первый же словарь, *распознавший* фрагмент, выдаёт одну или несколько представляющих его *лексем*. Например, `rats` превращается в `rat`, так как один из словарей понимает, что слово `rats` — это слово `rat` во множественном числе. Некоторое слова распознаются как *стоп-слова* (Подраздел 12.6.1) и игнорируются как слова, фигурирующие в тексте настолько часто, что искать их бессмысленно. В нашем примере это `a`, `on` и `it`. Если фрагмент не воспринимается ни одним словарём из списка, он так же игнорируется. В данном примере это происходит со знаком препинания `-`, так как с таким типом фрагмента (`символы-разделители`) не связан никакой словарь и значит такие фрагменты никогда не будут индексироваться. Выбор анализатора, словарей и индексируемых типов фрагментов определяется конфигурацией текстового поиска (Раздел 12.7). В одной базе данных можно использовать разные конфигурации, в том числе, предопределённые конфигурации для разных языков. В нашем примере мы использовали конфигурацию по умолчанию для английского языка — `english`.

Для назначения элементам `tsvector` разных *весов* используется функция `setweight`. Вес элемента задаётся буквой `A`, `B`, `C` или `D`. Обычно это применяется для обозначения важности слов в разных

частях документа, например в заголовке или в теле документа. Затем эта информация может использоваться при ранжировании результатов поиска.

Так как to_tsvector(NULL) вернёт NULL, мы советуем использовать coalesce везде, где соответствующее поле может быть NULL. Создавать tsvector из структурированного документа рекомендуется так:

```
UPDATE tt SET ti =
    setweight(to_tsvector(coalesce(title,'')), 'A')    ||
    setweight(to_tsvector(coalesce(keyword,'')), 'B')  ||
    setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
    setweight(to_tsvector(coalesce(body,'')), 'D');
```

Здесь мы использовали setweight для пометки происхождения каждой лексемы в сформированных значениях tsvector и объединили помеченные значения с помощью оператора конкатенации типов tsvector ||. (Подробнее эти операции рассматриваются в Подразделе 12.4.1.)

12.3.2. Разбор запросов

Postgres Pro предоставляет функции to_tsquery, plainto_tsquery и phraseto_tsquery для приведения запроса к типу данных tsquery. Функция to_tsquery даёт больше возможностей, чем plainto_tsquery и phraseto_tsquery, но более строга к входному запросу.

to_tsquery([*конфигурация* regconfig,] *текст_запроса* text) returns tsquery

to_tsquery создаёт значение tsquery из *текста_запроса*, который может состоять из простых фрагментов, разделённых логическими операторами tsquery: & (И), | (ИЛИ), ! (НЕ) и <-> (ПРЕДШЕСТВУЕТ), возможно, сгруппированных скобками. Другими словами, входное значение для to_tsquery должно уже соответствовать общим правилам для значений tsquery, описанным в Подразделе 8.11.2. Различие их состоит в том, что во вводимом в tsquery значении фрагменты воспринимаются буквально, тогда как to_tsquery нормализует фрагменты, приводя их к лексемам, используя явно указанную или подразумеваемую конфигурацию, и отбрасывая стоп-слова. Например:

```
SELECT to_tsquery('english', 'The & Fat & Rats');
  to_tsquery
---------------
 'fat' & 'rat'
```

Как и при вводе значения tsquery, для каждой лексемы можно задать вес(а), чтобы при поиске можно было выбрать из tsvector только лексемы с заданными весами. Например:

```
SELECT to_tsquery('english', 'Fat | Rats:AB');
    to_tsquery
------------------
 'fat' | 'rat':AB
```

К лексеме также можно добавить *, определив таким образом условие поиска по префиксу:

```
SELECT to_tsquery('supern:*A & star:A*B');
        to_tsquery
--------------------------
 'supern':*A & 'star':*AB
```

Такая лексема будет соответствовать любому слову в tsvector, начинающемуся с данной подстроки.

to_tsquery может также принимать фразы в апострофах. Это полезно в основном когда конфигурация включает тезаурус, который может обрабатывать такие фразы. В показанном ниже примере предполагается, что тезаурус содержит правило supernovae stars : sn:

```
SELECT to_tsquery('''supernovae stars'' & !crab');
  to_tsquery
```

```
---------------
 'sn' & !'crab'
```

Если убрать эти апострофы, `to_tsquery` не примет фрагменты, не разделённые операторами И, ИЛИ и ПРЕДШЕСТВУЕТ, и выдаст синтаксическую ошибку.

`plainto_tsquery([`*`конфигурация`* `regconfig,] `*`текст_запроса`* `text) returns tsquery`

`plainto_tsquery` преобразует неформатированный *`текст_запроса`* в значение `tsquery`. Текст разбирается и нормализуется подобно тому, как это делает `to_tsvector`, а затем между оставшимися словами вставляются операторы `&` (И) типа `tsquery`.

Пример:

```
SELECT plainto_tsquery('english', 'The Fat Rats');
 plainto_tsquery
-----------------
 'fat' & 'rat'
```

Заметьте, что `plainto_tsquery` не распознает во входной строке операторы `tsquery`, метки весов или обозначения префиксов:

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');
   plainto_tsquery
---------------------
 'fat' & 'rat' & 'c'
```

В данном случае все знаки пунктуации были отброшены как символы-разделители.

`phraseto_tsquery([`*`конфигурация`* `regconfig,] `*`текст_запроса`* `text) returns tsquery`

`phraseto_tsquery` ведёт себя подобно `plainto_tsquery`, за исключением того, что она вставляет между оставшимися словами оператор `<->` (ПРЕДШЕСТВУЕТ) вместо оператора `&` (И). Кроме того, стоп-слова не просто отбрасываются, а подсчитываются, и вместо операторов `<->` используются операторы `<`*`N`*`>` с подсчитанным числом. Эта функция полезна при поиске точных последовательностей лексем, так как операторы ПРЕДШЕСТВУЕТ проверяют не только наличие всех лексем, но и их порядок.

Пример:

```
SELECT phraseto_tsquery('english', 'The Fat Rats');
 phraseto_tsquery
------------------
 'fat' <-> 'rat'
```

Как и `plainto_tsquery`, функция `phraseto_tsquery` не распознает во входной строке операторы типа `tsquery`, метки весов или обозначения префиксов:

```
SELECT phraseto_tsquery('english', 'The Fat & Rats:C');
      phraseto_tsquery
-----------------------------
 'fat' <-> 'rat' <-> 'c'
```

12.3.3. Ранжирование результатов поиска

Ранжирование документов можно представить как попытку оценить, насколько они релевантны заданному запросу и отсортировать их так, чтобы наиболее релевантные выводились первыми. В Postgres Pro встроены две функции ранжирования, принимающие во внимание лексическую, позиционную и структурную информацию; то есть, они учитывают, насколько часто и насколько близко встречаются в документе ключевые слова и какова важность содержащей их части документа. Однако само понятие релевантности довольно размытое и во многом определяется приложением. Приложения могут использовать для ранжирования и другую

информацию, например, время изменения документа. Встроенные функции ранжирования можно рассматривать лишь как примеры реализации. Для своих конкретных задач вы можете разработать собственные функции ранжирования и/или учесть при обработке их результатов дополнительные факторы.

Ниже описаны две встроенные функции ранжирования:

`ts_rank([`*`веса`* `float4[],]` *`вектор`* `tsvector,` *`запрос`* `tsquery [,` *`нормализация`* `integer]) returns float4`

Ранжирует векторы по частоте найденных лексем.

`ts_rank_cd([`*`веса`* `float4[],]` *`вектор`* `tsvector,` *`запрос`* `tsquery [,` *`нормализация`* `integer]) returns float4`

Эта функция вычисляет *плотность покрытия* для данного вектора документа и запроса, используя метод, разработанный Кларком, Кормаком и Тадхоуп и описанный в статье «Relevance Ranking for One to Three Term Queries» в журнале «Information Processing and Management» в 1999 г. Плотность покрытия вычисляется подобно рангу `ts_rank`, но в расчёт берётся ещё и близость соответствующих лексем друг к другу.

Для вычисления результата этой функции требуется информация о позиции лексем. Поэтому она игнорируют «очищенные» от этой информации лексемы в `tsvector`. Если во входных данных нет неочищенных лексем, результат будет равен нулю. (За дополнительными сведениями о функции `strip` и позиционной информации в данных `tsvector` обратитесь к Подразделу 12.4.1.)

Для обеих этих функций аргумент *`веса`* позволяет придать больший или меньший вес словам, в зависимости от их меток. В передаваемом массиве весов определяется, насколько весома каждая категория слов, в следующем порядке:

```
{вес D, вес C, вес B, вес A}
```

Если этот аргумент опускается, подразумеваются следующие значения:

```
{0.1, 0.2, 0.4, 1.0}
```

Обычно весами выделяются слова из особых областей документа, например из заголовка или краткого введения, с тем, чтобы эти слова считались более и менее значимыми, чем слова в основном тексте документа.

Так как вероятность найти ключевые слова увеличивается с размером документа, при ранжировании имеет смысл учитывать его, чтобы, например, документ с сотней слов, содержащий пять вхождений искомых слов, считался более релевантным, чем документ с тысячей слов и теми же пятью вхождениями. Обе функции ранжирования принимают целочисленный параметр *`нормализации`*, определяющий, как ранг документа будет зависеть от его размера. Этот параметр представляет собой битовую маску и управляет несколькими режимами: вы можете включить сразу несколько режимов, объединив значения оператором `|` (например так: `2|4`).

- 0 (по умолчанию): длина документа не учитывается
- 1: ранг документа делится на 1 + логарифм длины документа
- 2: ранг документа делится на его длину
- 4: ранг документа делится на среднее гармоническое расстояние между блоками (это реализовано только в `ts_rank_cd`)
- 8: ранг документа делится на число уникальных слов в документе
- 16: ранг документа делится на 1 + логарифм числа уникальных слов в документе
- 32: ранг делится своё же значение + 1

Если включены несколько флагов, соответствующие операции выполняются в показанном порядке.

Важно заметить, что функции ранжирования не используют никакую внешнюю информацию, так что добиться нормализации до 1% или 100% невозможно, хотя иногда это желательно. Применив

параметр 32 (`rank/(rank+1)`), можно свести все ранги к диапазону 0..1, но это изменение будет лишь косметическим, на порядке сортировки результатов это не отразится.

В данном примере выбираются десять найденных документов с максимальным рангом:

```
SELECT title, ts_rank_cd(textsearch, query) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |   rank
-----------------------------------------------+----------
 Neutrinos in the Sun                          |      3.1
 The Sudbury Neutrino Detector                 |      2.4
 A MACHO View of Galactic Dark Matter          |  2.01317
 Hot Gas and Dark Matter                       |  1.91171
 The Virgo Cluster: Hot Plasma and Dark Matter |  1.90953
 Rafting for Solar Neutrinos                   |      1.9
 NGC 4650A: Strange Galaxy and Dark Matter     |  1.85774
 Hot Gas and Dark Matter                       |   1.6123
 Ice Fishing for Cosmic Neutrinos              |      1.6
 Weak Lensing Distorts the Universe            | 0.818218
```

Тот же пример с нормализованным рангом:

```
SELECT title, ts_rank_cd(textsearch, query, 32 /* rank/(rank+1) */ ) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE  query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |        rank
-----------------------------------------------+-------------------
 Neutrinos in the Sun                          | 0.756097569485493
 The Sudbury Neutrino Detector                 | 0.705882361190954
 A MACHO View of Galactic Dark Matter          | 0.668123210574724
 Hot Gas and Dark Matter                       |  0.65655958650282
 The Virgo Cluster: Hot Plasma and Dark Matter | 0.656301290640973
 Rafting for Solar Neutrinos                   | 0.655172410958162
 NGC 4650A: Strange Galaxy and Dark Matter     | 0.650072921219637
 Hot Gas and Dark Matter                       | 0.617195790024749
 Ice Fishing for Cosmic Neutrinos              | 0.615384618911517
 Weak Lensing Distorts the Universe            | 0.450010798361481
```

Ранжирование может быть довольно дорогостоящей операцией, так как для вычисления ранга необходимо прочитать `tsvector` каждого подходящего документа и это займёт значительное время, если придётся обращаться к диску. К сожалению, избежать этого вряд ли возможно, так как на практике по многим запросам выдаётся большое количество результатов.

12.3.4. Выделение результатов

Представляя результаты поиска, в идеале нужно выделять часть документа и показывать, как он связан с запросом. Обычно поисковые системы показывают фрагменты документа с отмеченными искомыми словами. В Postgres Pro для реализации этой возможности представлена функция `ts_headline`.

```
ts_headline([конфигурация regconfig,] документ text, запрос tsquery [, параметры text])
  returns text
```

`ts_headline` принимает документ вместе с запросом и возвращает выдержку из документа, в которой выделяются слова из запроса. Применяемую для разбора документа конфигурацию

можно указать в параметре *config*; если этот параметр опущен, применяется конфигурация default_text_search_config.

Если в параметрах передаётся строка *options*, она должна состоять из списка разделённых запятыми пар *параметр=значение*. Параметры могут быть следующими:

- MaxWords, MinWords (целочисленные): эти числа определяют нижний и верхний предел размера выдержки. Значения по умолчанию: 35 и 15, соответственно.
- ShortWord (целочисленный): слова такой длины или короче в начале и конце выдержки будут отбрасываться, за исключением искомых слов. Значение по умолчанию, равное 3, исключает распространённые английские артикли.
- HighlightAll (логический): при значении true выдержкой будет весь документ, и три предыдущие параметра игнорируются. Значение по умолчанию: false.
- MaxFragments (целочисленный): максимальное число выводимых текстовых фрагментов. Значение по умолчанию, равное нулю, выбирает режим создания выдержек без фрагментов. При положительном значении выбирается режим с фрагментами (см. ниже).
- StartSel, StopSel: строки, которые будут разграничивать слова запроса в документе, выделяя их среди остальных. Если эти строки содержат пробелы или запятые, их нужно заключить в кавычки.
- FragmentDelimiter (строка): в случае, когда фрагментов несколько, они будут разделяться указанной строкой. Значение по умолчанию: « ... ».

Имена этих параметров распознаются без учёта регистра. Значения, содержащие пробелы или запятые, должны заключаться в двойные кавычки.

В режиме формирования выдержек без фрагментов ts_headline находит вхождения слов заданного *запроса* и возвращает одно из вхождений, предпочитая те, что содержат как можно больше слов из запроса в пределах допустимого размера выдержки. В режиме с фрагментами ts_headline находит вхождения слов запроса и разделяет эти вхождения на «фрагменты», состоящие не более чем из MaxWords слов, предпочитая те, что содержат больше искомых слов, а затем может «растянуть» фрагменты, добавив в них соседние слова. Второй режим полезнее, когда слова запроса находятся не рядом, а разбросаны по документу, или когда желательно увидеть сразу несколько вхождений. В случаях, когда соответствие запросу найти не удаётся, в обоих режимах возвращаются первые MinWords слов из документа.

Пример использования:

```
SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('english', 'query & similarity'));
                        ts_headline
------------------------------------------------------------
 containing given <b>query</b> terms                       +
 and return them in order of their <b>similarity</b> to the+
 <b>query</b>.

SELECT ts_headline('english',
  'Search terms may occur
many times in a document,
requiring ranking of the search matches to decide which
occurrences to display in the result.',
  to_tsquery('english', 'search & term'),
  'MaxFragments=10, MaxWords=7, MinWords=3, StartSel=<<, StopSel=>>');
                        ts_headline
------------------------------------------------------------
 <<Search>> <<terms>> may occur                            +
```

```
 many times ... ranking of the <<search>> matches to decide
```

Функция `ts_headline` работает с оригинальным документом, а не его сжатым представлением `tsvector`, так что она может быть медленной и использовать её следует осмотрительно.

12.4. Дополнительные возможности

В этом разделе описываются дополнительные функции и операторы, которые могут быть полезны при поиске текста.

12.4.1. Обработка документов

В Подразделе 12.3.1 показывалось, как обычные текстовые документы можно преобразовать в значения `tsvector`. Postgres Pro предлагает также набор функций и операторов для обработки документов, уже представленных в формате `tsvector`.

`tsvector || tsvector`

Оператор конкатенации значений `tsvector` возвращает вектор, объединяющий лексемы и позиционную информацию двух векторов, переданных ему в аргументах. В полученном результате сохраняются позиции и метки весов. При этом позиции в векторе справа сдвигаются на максимальное значение позиции в векторе слева, что почти равносильно применению `to_tsvector` к результату конкатенации двух исходных строк документов. (Почти, потому что стоп-слова, исключаемые в конце левого аргумента, при конкатенации исходных строк влияют на позиции лексем в правой части, а при конкатенации `tsvector` — нет.)

Преимущество же конкатенации документов в векторной форме по сравнению с конкатенацией текста до вызова `to_tsvector` заключается в том, что так можно разбирать разные части документа, применяя разные конфигурации. И так как функция `setweight` помечает все лексемы данного вектора одинаково, разбирать текст и выполнять `setweight` нужно до объединения разных частей документа с подразумеваемым разным весом.

`setweight(`*`вектор`* `tsvector,` *`вес`* `"char") returns tsvector`

`setweight` возвращает копию входного вектора, помечая в ней каждую позицию заданным *`весом`*, меткой `A`, `B`, `C` или `D`. (Метка `D` по умолчанию назначается всем векторам, так что при выводе она опускается.) Эти метки сохраняются при конкатенации векторов, что позволяет придавать разные веса словам из разных частей документа и, как следствие, ранжировать их по-разному.

Заметьте, что веса назначаются *позициям*, а не *лексемам*. Если входной вектор очищен от позиционной информации, `setweight` не делает ничего.

`length(`*`вектор`* `tsvector) returns integer`

Возвращает число лексем, сохранённых в векторе.

`strip(`*`вектор`* `tsvector) returns tsvector`

Возвращает вектор с теми же лексемами, что и в данном, но без информации о позиции и весе. Очищенный вектор обычно оказывается намного меньше исходного, но при этом и менее полезным. С очищенными векторами хуже работает ранжирование, а также оператор `<->` (ПРЕДШЕСТВУЕТ) типа `tsquery` никогда не найдёт соответствие в них, так как не сможет определить расстояние между вхождениями лексем.

Полный список связанных с `tsvector` функций приведён в Таблице 9.41.

12.4.2. Обработка запросов

В Подразделе 12.3.2 показывалось, как обычные текстовые запросы можно преобразовывать в значения `tsquery`. Postgres Pro предлагает также набор функций и операторов для обработки запросов, уже представленных в формате `tsquery`.

`tsquery && tsquery`

Возвращает логическое произведение (AND) двух данных запросов.

`tsquery || tsquery`

Возвращает логическое объединение (OR) двух данных запросов.

`!! tsquery`

Возвращает логическое отрицание (NOT) данного запроса.

`tsquery <-> tsquery`

Возвращает запрос, который ищет соответствие первому данному запросу, за которым следует соответствие второму данному запросу, с применением оператора `<->` (ПРЕДШЕСТВУЕТ) типа `tsquery`. Например:

```
SELECT to_tsquery('fat') <-> to_tsquery('cat | rat');
          ?column?
----------------------------
 'fat' <-> ( 'cat' | 'rat' )
```

`tsquery_phrase(`*`запрос1`* `tsquery, `*`запрос2`* `tsquery [, `*`расстояние`* `integer ]) returns tsquery`

Возвращает запрос, который ищет соответствие первому данному запросу, за которым следует соответствие второму данному запросу (точное число лексем между ними задаётся параметром *`расстояние`*), с применением оператора `<`*`N`*`>` типа `tsquery`. Например:

```
SELECT tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10);
  tsquery_phrase
------------------
 'fat' <10> 'cat'
```

`numnode(`*`запрос`* `tsquery) returns integer`

Возвращает число узлов (лексем и операторов) в значении `tsquery`. Эта функция помогает определить, имеет ли смысл *`запрос`* (тогда её результат > 0) или он содержит только стоп-слова (тогда она возвращает 0). Примеры:

```
 SELECT numnode(plainto_tsquery('the any'));
ЗАМЕЧАНИЕ:  запрос поиска текста игнорируется, так как содержит
 только стоп-слова или не содержит лексем
 numnode
---------
       0

SELECT numnode('foo & bar'::tsquery);
 numnode
---------
       3
```

`querytree(`*`запрос`* `tsquery) returns text`

Возвращает часть `tsquery`, которую можно использовать для поиска по индексу. Эта функция помогает выявить неиндексируемые запросы, например, такие, которые содержат только стоп-слова или условия отрицания. Например:

```
SELECT querytree(to_tsquery('!defined'));
 querytree
-----------
```

12.4.2.1. Перезапись запросов

Семейство запросов `ts_rewrite` ищет в данном `tsquery` вхождения целевого подзапроса и заменяет каждое вхождение указанной подстановкой. По сути эта операция похожа на замену подстроки в строке, только рассчитана на работу с `tsquery`. Сочетание целевого подзапроса с подстановкой можно считать *правилом перезаписи запроса*. Набор таких правил перезаписи может быть очень полезен при поиске. Например, вы можете улучшить результаты, добавив синонимы (например, `big  apple`, `nyc` и `gotham` для `new  york`) или сузить область поиска, чтобы нацелить пользователя на некоторую область. Это в некотором смысле пересекается с функциональностью тезаурусов (Подраздел 12.6.4). Однако при таком подходе вы можете изменять правила перезаписи «на лету», тогда как при обновлении тезауруса необходима переиндексация.

`ts_rewrite (`*`запрос`* `tsquery, `*`цель`* `tsquery, `*`замена`* `tsquery) returns tsquery`

Эта форма `ts_rewrite` просто применяет одно правило перезаписи: *`цель`* заменяется *`подстановкой`* везде, где она находится в *`запросе`*. Например:

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);
 ts_rewrite
------------
 'b' & 'c'
```

`ts_rewrite (`*`запрос`* `tsquery, `*`выборка`* `text) returns tsquery`

Эта форма `ts_rewrite` принимает начальный *`запрос`* и SQL-команду *`select`*, которая задаётся текстовой строкой. Команда *`select`* должна выдавать два столбца типа `tsquery`. Для каждой строки результата *`select`* вхождения первого столбца (цели) заменяются значениями второго столбца (подстановкой) в тексте *`запроса`*. Например:

```
CREATE TABLE aliases (t tsquery PRIMARY KEY, s tsquery);
INSERT INTO aliases VALUES('a', 'c');

SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases');
 ts_rewrite
------------
 'b' & 'c'
```

Заметьте, что когда таким способом применяются несколько правил перезаписи, порядок их применения может иметь значение, поэтому в исходном запросе следует добавить `ORDER BY` по какому-либо ключу.

Давайте рассмотрим практический пример на тему астрономии. Мы развернём запрос `supernovae`, используя правила перезаписи в таблице:

```
CREATE TABLE aliases (t tsquery primary key, s tsquery);
INSERT INTO aliases VALUES(to_tsquery('supernovae'),
  to_tsquery('supernovae|sn'));

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
           ts_rewrite
---------------------------------
 'crab' & ( 'supernova' | 'sn' )
```

Мы можем скорректировать правила перезаписи, просто изменив таблицу:

```
UPDATE aliases
SET s = to_tsquery('supernovae|sn & !nebulae')
WHERE t = to_tsquery('supernovae');

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
                 ts_rewrite
```

```
---------------------------------------------
 'crab' & ( 'supernova' | 'sn' & !'nebula' )
```

Перезапись может быть медленной, когда задано много правил перезаписи, так как соответствия будут проверяться для каждого правила. Чтобы отфильтровать явно неподходящие правила, можно использовать проверки включения для типа `tsquery`. В следующем примере выбираются только те правила, которые могут соответствовать исходному запросу:

```
SELECT ts_rewrite('a & b'::tsquery,
                  'SELECT t,s FROM aliases WHERE ''a & b''::tsquery @> t');
 ts_rewrite
------------
 'b' & 'c'
```

12.4.3. Триггеры для автоматического обновления

Когда представление документа в формате `tsvector` хранится в отдельном столбце, необходимо создать триггер, который будет обновлять его содержимое при изменении столбцов, из которых составляется исходный документ. Для этого можно использовать две встроенные триггерные функции или написать свои собственные.

```
tsvector_update_trigger(столбец_tsvector, имя_конфигурации, столбец_текста [, ...])
tsvector_update_trigger_column(столбец_tsvector, столбец_конфигурации,
столбец_текста [, ...])
```

Эти триггерные функции автоматически вычисляют значение для столбца `tsvector` из одного или нескольких текстовых столбцов с параметрами, указанными в команде `CREATE TRIGGER`. Пример их использования:

```
CREATE TABLE messages (
    title       text,
    body        text,
    tsv         tsvector
);

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE PROCEDURE
tsvector_update_trigger(tsv, 'pg_catalog.english', title, body);

INSERT INTO messages VALUES('title here', 'the body text is here');

SELECT * FROM messages;
   title    |         body          |            tsv
------------+-----------------------+----------------------------
 title here | the body text is here | 'bodi':4 'text':5 'titl':1

SELECT title, body FROM messages WHERE tsv @@ to_tsquery('title & body');
   title    |         body
------------+-----------------------
 title here | the body text is here
```

С таким триггером любое изменение в полях `title` или `body` будет автоматически отражаться в содержимом `tsv`, так что приложению не придётся заниматься этим.

Первым аргументом этих функций должно быть имя столбца `tsvector`, содержимое которого будет обновляться. Ещё один аргумент — конфигурация текстового поиска, которая будет использоваться для преобразования. Для `tsvector_update_trigger` имя конфигурации передаётся просто как второй аргумент триггера. Это имя должно быть определено полностью, чтобы поведение триггера не менялось при изменениях в пути поиска (`search_path`). Для `tsvector_update_trigger_column` во втором аргументе триггера передаётся имя другого столбца

таблицы, который должен иметь тип `regconfig`. Это позволяет использовать разные конфигурации для разных строк. В оставшихся аргументах передаются имена текстовых столбцов (типа `text`, `varchar` или `char`). Их содержимое будет включено в документ в заданном порядке. При этом значения NULL будут пропущены (а другие столбцы будут индексироваться).

Ограничение этих встроенных триггеров заключается в том, что они обрабатывают все столбцы одинаково. Чтобы столбцы обрабатывались по-разному, например для текста заголовка задавался не тот же вес, что для тела документа, потребуется разработать свой триггер. К примеру, так это можно сделать на языке PL/pgSQL:

```
CREATE FUNCTION messages_trigger() RETURNS trigger AS $$
begin
  new.tsv :=
     setweight(to_tsvector('pg_catalog.english', coalesce(new.title,'')),
       'A') ||
     setweight(to_tsvector('pg_catalog.english', coalesce(new.body,'')),
       'D');
  return new;
end
$$ LANGUAGE plpgsql;

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
    ON messages FOR EACH ROW EXECUTE PROCEDURE messages_trigger();
```

Помните, что, создавая значения `tsvector` в триггерах, важно явно указывать имя конфигурации, чтобы содержимое столбца не зависело от изменений `default_text_search_config`. В противном случае могут возникнуть проблемы, например результаты поиска изменятся после выгрузки и восстановления данных.

12.4.4. Сбор статистики по документу

Функция `ts_stat` может быть полезна для проверки конфигурации и нахождения возможных стоп-слов.

```
ts_stat(sql_запрос text, [веса text,]
        OUT слово text, OUT число_док integer,
        OUT число_вхожд integer) returns setof record
```

Здесь *sql_запрос* — текстовая строка, содержащая SQL-запрос, который должен возвращать один столбец `tsvector`. Функция `ts_stat` выполняет запрос и возвращает статистику по каждой отдельной лексеме (слову), содержащейся в данных `tsvector`. Её результат представляется в столбцах

- *слово* `text` — значение лексемы
- *число_док* `integer` — число документов (значений `tsvector`), в которых встретилось слово
- *число_вхожд* `integer` — общее число вхождений слова

Если передаётся параметр *weights*, то подсчитываются только вхождения с указанными в нём весами.

Например, найти десять наиболее часто используемых слов в коллекции документов можно так:

```
SELECT * FROM ts_stat('SELECT vector FROM apod')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

Следующий запрос возвращает тоже десять слов, но при выборе их учитываются только вхождения с весами `A` или `B`:

```
SELECT * FROM ts_stat('SELECT vector FROM apod', 'ab')
ORDER BY nentry DESC, ndoc DESC, word
```

```
LIMIT 10;
```

12.5. Анализаторы

Задача анализаторов текста — разделить текст документа на *фрагменты* и присвоить каждому из них тип из набора, определённого в самом анализаторе. Заметьте, что анализаторы не меняют текст — они просто выдают позиции предполагаемых слов. Вследствие такой ограниченности их функций, собственные специфические анализаторы бывают нужны гораздо реже, чем собственные словари. В настоящее время в Postgres Pro есть только один встроенный анализатор, который может быть полезен для широкого круга приложений.

Этот встроенный анализатор называется `pg_catalog.default`. Он распознаёт 23 типа фрагментов, перечисленные в Таблице 12.1.

Таблица 12.1. Типы фрагментов, выделяемых стандартным анализатором

Псевдоним	Описание	Пример
`asciiword`	Слово только из букв ASCII	`elephant`
`word`	Слово из любых букв	`mañana`
`numword`	Слово из букв и цифр	`beta1`
`asciihword`	Слово только из букв ASCII с дефисами	`up-to-date`
`hword`	Слово из любых букв с дефисами	`lógico-matemática`
`numhword`	Слово из букв и цифр с дефисами	`postgresql-beta1`
`hword_asciipart`	Часть слова с дефисами, только из букв ASCII	`postgresql` в словосочетании `postgresql-beta1`
`hword_part`	Часть слова с дефисами, из любых букв	`lógico` или `matemática` в словосочетании `lógico-matemática`
`hword_numpart`	Часть слова с дефисами, из букв и цифр	`beta1` в словосочетании `postgresql-beta1`
`email`	Адрес электронной почты	`foo@example.com`
`protocol`	Префикс протокола	`http://`
`url`	URL	`example.com/stuff/index.html`
`host`	Имя узла	`example.com`
`url_path`	Путь в адресе URL	`/stuff/index.html`, как часть URL
`file`	Путь или имя файла	`/usr/local/foo.txt`, если не является частью URL
`sfloat`	Научная запись числа	`-1.234e56`
`float`	Десятичная запись числа	`-1.234`
`int`	Целое со знаком	`-1234`
`uint`	Целое без знака	`1234`
`version`	Номер версии	`8.3.0`
`tag`	Тег XML	`<a href="dictionaries.html">`
`entity`	Сущность XML	`&`

Псевдоним	Описание	Пример
blank	Символы-разделители	(любые пробельные символы или знаки препинания, не попавшие в другие категории)

Примечание

Понятие «буквы» анализатор определяет исходя из локали, заданной для базы данных, в частности параметра lc_ctype. Слова, содержащие только буквы из ASCII (латинские буквы), распознаются как фрагменты отдельного типа, так как иногда бывает полезно выделить их. Для многих европейских языков типы фрагментов word и asciiword можно воспринимать как синонимы.

email принимает не все символы, которые считаются допустимыми по стандарту RFC 5322. В частности, имя почтового ящика помимо алфавитно-цифровых символов может содержать только точку, минус и подчёркивание.

Анализатор может выделить в одном тексте несколько перекрывающихся фрагментов. Например, слово с дефисом будет выдано как целое составное слово и по частям:

```
SELECT alias, description, token FROM ts_debug('foo-bar-beta1');
      alias      |               description                |     token
-----------------+------------------------------------------+--------------
 numhword        | Hyphenated word, letters and digits      | foo-bar-beta1
 hword_asciipart | Hyphenated word part, all ASCII          | foo
 blank           | Space symbols                            | -
 hword_asciipart | Hyphenated word part, all ASCII          | bar
 blank           | Space symbols                            | -
 hword_numpart   | Hyphenated word part, letters and digits | beta1
```

Это поведение считается желательным, так как это позволяет находить при последующем поиске и всё слово целиком, и его части. Ещё один показательный пример:

```
SELECT alias, description, token
FROM ts_debug('http://example.com/stuff/index.html');
  alias   |  description  |            token
----------+---------------+------------------------------
 protocol | Protocol head | http://
 url      | URL           | example.com/stuff/index.html
 host     | Host          | example.com
 url_path | URL path      | /stuff/index.html
```

12.6. Словари

Словари полнотекстового поиска предназначены для исключения *стоп-слов* (слов, которые не должны учитываться при поиске) и *нормализации* слов, чтобы разные словоформы считались совпадающими. Успешно нормализованное слово называется *лексемой*. Нормализация и исключение стоп-слов не только улучшает качество поиска, но и уменьшает размер представления документа в формате tsvector, и, как следствие, увеличивает быстродействие. Нормализация не всегда имеет лингвистический смысл, обычно она зависит от требований приложения.

Несколько примеров нормализации:

- Лингвистическая нормализация — словари Ispell пытаются свести слова на входе к нормализованной форме, а стеммеры убирают окончания слов
- Адреса URL могут быть канонизированы, чтобы например следующие адреса считались одинаковыми:
  - http://www.pgsql.ru/db/mw/index.html

  - http://www.pgsql.ru/db/mw/
  - http://www.pgsql.ru/db/../db/mw/index.html
- Названия цветов могут быть заменены их шестнадцатеричными значениями, например `red, green, blue, magenta -> FF0000, 00FF00, 0000FF, FF00FF`
- При индексировании чисел можно отбросить цифры в дробной части для сокращения множества всевозможных чисел, чтобы например *3.14*159265359, *3.14*15926 и *3.14* стали одинаковыми после нормализации, при которой после точки останутся только две цифры.

Словарь — это программа, которая принимает на вход фрагмент и возвращает:

- массив лексем, если входной фрагмент известен в словаре (заметьте, один фрагмент может породить несколько лексем)
- одну лексему с установленным флагом `TSL_FILTER` для замены исходного фрагмента новым, чтобы следующие словари работали с новым вариантом (словарь, который делает это, называется *фильтрующим словарём*)
- пустой массив, если словарь воспринимает этот фрагмент, но считает его стоп-словом
- `NULL`, если словарь не воспринимает полученный фрагмент

В Postgres Pro встроены стандартные словари для многих языков. Есть также несколько предопределённых шаблонов, на основании которых можно создавать новые словари с изменёнными параметрами. Все эти шаблоны описаны ниже. Если же ни один из них не подходит, можно создать и свои собственные шаблоны. Соответствующие примеры можно найти в каталоге `contrib/` инсталляции Postgres Pro.

Конфигурация текстового поиска связывает анализатор с набором словарей, которые будут обрабатывать выделенные им фрагменты. Для каждого типа фрагментов, выданных анализатором, в конфигурации задаётся отдельный список словарей. Найденный анализатором фрагмент проходит через все словари по порядку, пока какой-либо словарь не увидит в нём знакомое для него слово. Если он окажется стоп-словом или его не распознает ни один словарь, этот фрагмент не будет учитываться при индексации и поиске. Обычно результат определяет первый же словарь, который возвращает не `NULL`, и остальные словари уже не проверяются; однако фильтрующий словарь может заменить полученное слово другим, которое и будет передано следующим словарям.

Общее правило настройки списка словарей заключается в том, чтобы поставить наиболее частные и специфические словари в начале, затем перечислить более общие и закончить самым общим словарём, например стеммером Snowball или словарём `simple`, который распознаёт всё. Например, для поиска по теме астрономии (конфигурация `astro_en`) тип фрагментов `asciiword` (слово из букв ASCII) можно связать со словарём синонимов астрономических терминов, затем с обычным английским словарём и наконец со стеммером английских окончаний Snowball:

```
ALTER TEXT SEARCH CONFIGURATION astro_en
    ADD MAPPING FOR asciiword WITH astrosyn, english_ispell, english_stem;
```

Фильтрующий словарь можно включить в любом месте списка, кроме конца, где он будет бесполезен. Фильтрующие словари бывают полезны для частичной нормализации слов и упрощения задачи следующих словарей. Например, фильтрующий словарь может удалить из текста диакритические знаки, как это делает модуль unaccent.

12.6.1. Стоп-слова

Стоп-словами называются слова, которые встречаются очень часто, практически в каждом документе, и поэтому не имеют различительной ценности. Таким образом, при полнотекстовом поиске их можно игнорировать. Например, в каждом английском тексте содержатся артикли `a` и `the`, так что хранить их в индексе бессмысленно. Однако стоп-слова влияют на позиции лексем в значении `tsvector`, от чего, в свою очередь, зависит ранжирование:

```
SELECT to_tsvector('english', 'in the list of stop words');
        to_tsvector
----------------------------
```

```
 'list':3 'stop':5 'word':6
```

В результате отсутствуют позиции 1,2,4, потому что фрагменты в этих позициях оказались стоп-словами. Ранги, вычисленные для документов со стоп-словами и без них, могут значительно различаться:

```
SELECT ts_rank_cd (to_tsvector('english', 'in the list of stop words'),
  to_tsquery('list & stop'));
 ts_rank_cd
------------
       0.05

SELECT ts_rank_cd (to_tsvector('english', 'list stop words'),
  to_tsquery('list & stop'));
 ts_rank_cd
------------
        0.1
```

Как именно обрабатывать стоп-слова, определяет сам словарь. Например, словари `ispell` сначала нормализуют слова, а затем просматривают список стоп-слов, тогда как стеммеры `Snowball` просматривают свой список стоп-слов в первую очередь. Это различие в поведении объясняется стремлением уменьшить шум.

12.6.2. Простой словарь

Работа шаблона словарей `simple` сводится к преобразованию входного фрагмента в нижний регистр и проверки результата по файлу со списком стоп-слов. Если это слово находится в файле, словарь возвращает пустой массив и фрагмент исключается из дальнейшего рассмотрения. В противном случае словарь возвращает в качестве нормализованной лексемы слово в нижнем регистре. Этот словарь можно настроить и так, чтобы все слова, кроме стоп-слов, считались неопознанными и передавались следующему словарю в списке.

Определить словарь на основе шаблона `simple` можно так:

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (
    TEMPLATE = pg_catalog.simple,
    STOPWORDS = english
);
```

Здесь `english` — базовое имя файла со стоп-словами. Полным именем файла будет `$SHAREDIR/tsearch_data/english.stop`, где `$SHAREDIR` указывает на каталог с общими данными Postgres Pro, часто это `/usr/local/share/postgresql` (точно узнать его можно с помощью команды `pg_config --sharedir`). Этот текстовый файл должен содержать просто список слов, по одному слову в строке. Пустые строки и окружающие пробелы игнорируются, все символы переводятся в нижний регистр и на этом обработка файла заканчивается.

Теперь мы можем проверить наш словарь:

```
SELECT ts_lexize('public.simple_dict', 'YeS');
 ts_lexize
-----------
 {yes}

SELECT ts_lexize('public.simple_dict', 'The');
 ts_lexize
-----------
 {}
```

Мы также можем настроить словарь так, чтобы он возвращал `NULL` вместо слова в нижнем регистре, если оно не находится в файле стоп-слов. Для этого нужно присвоить параметру `Accept` значение `false`. Продолжая наш пример:

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );

SELECT ts_lexize('public.simple_dict', 'YeS');
 ts_lexize
-----------


SELECT ts_lexize('public.simple_dict', 'The');
 ts_lexize
-----------
 {}
```

Со значением `Accept = true` (по умолчанию) словарь `simple` имеет смысл включать только в конце списка словарей, так как он никогда не передаст фрагмент следующему словарю. И напротив, `Accept = false` имеет смысл, только если за ним следует ещё минимум один словарь.

> **Внимание**
>
> Большинство словарей работают с дополнительными файлами, например, файлами стоп-слов. Содержимое этих файлов *должно* иметь кодировку UTF-8. Если база данных работает в другой кодировке, они будут переведены в неё, когда сервер будет загружать их.

> **Внимание**
>
> Обычно в рамках одного сеанса дополнительный файл словаря загружается только один раз, при первом использовании. Если же вы измените его и захотите, чтобы существующие сеансы работали с новым содержимым, выполните для этого словаря команду `ALTER TEXT SEARCH DICTIONARY`. Это обновление словаря может быть «фиктивным», фактически не меняющим значения никаких параметров.

12.6.3. Словарь синонимов

Этот шаблон словарей используется для создания словарей, заменяющих слова синонимами. Словосочетания такие словари не поддерживают (используйте для этого тезаурус (Подраздел 12.6.4)). Словарь синонимов может помочь в преодолении лингвистических проблем, например, не дать стеммеру английского уменьшить слово «Paris» до «pari». Для этого достаточно поместить в словарь синонимов строку `Paris paris` и поставить этот словарь перед словарём `english_stem`. Например:

```
SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token|  dictionaries |  dictionary | lexemes
-----------+-----------------+------+---------------+-------------+--------
 asciiword| Word, all ASCII| Paris| {english_stem}| english_stem| {pari}

CREATE TEXT SEARCH DICTIONARY my_synonym (
    TEMPLATE = synonym,
    SYNONYMS = my_synonyms
);

ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR asciiword
    WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token| dictionaries | dictionary| lexemes
```

```
----------+---------------+------+-------------+-----------+--------
 asciiword| Word, all ASCII| Paris| {my_synonym, | my_synonym| {paris}
          |               |      | english_stem}|           |
```

Шаблон `synonym` принимает единственный параметр, `SYNONYMS`, в котором задаётся базовое имя его файла конфигурации — в данном примере это `my_synonyms`. Полным именем файла будет `$SHAREDIR/tsearch_data/my_synonyms.syn` (где `$SHAREDIR` указывает на каталог общих данных Postgres Pro). Содержимое этого файла должны составлять строки с двумя словами в каждой (первое — заменяемое слово, а второе — его синоним), разделёнными пробелами. Пустые строки и окружающие пробелы при разборе этого файла игнорируются.

Шаблон `synonym` также принимает необязательный параметр `CaseSensitive`, который по умолчанию имеет значение `false`. Когда `CaseSensitive` равен `false`, слова в файле синонимов переводятся в нижний регистр, вместе с проверяемыми фрагментами. Если же он не равен `true`, регистр слов в файле и проверяемых фрагментов не меняются, они сравниваются «как есть».

В конце синонима в этом файле можно добавить звёздочку (*), тогда этот синоним будет рассматриваться как префикс. Эта звёздочка будет игнорироваться в `to_tsvector()`, но `to_tsquery()` изменит результат, добавив в него маркер сопоставления префикса (см. Подраздел 12.3.2). Например, предположим, что файл `$SHAREDIR/tsearch_data/synonym_sample.syn` имеет следующее содержание:

```
postgres        pgsql
postgresql      pgsql
postgre pgsql
gogle   googl
indices index*
```

С ним мы получим такие результаты:

```
mydb=# CREATE TEXT SEARCH DICTIONARY syn (template=synonym, synonyms='synonym_sample');
mydb=# SELECT ts_lexize('syn', 'indices');
 ts_lexize
-----------
 {index}
(1 row)

mydb=# CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);
mydb=# ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH syn;
mydb=# SELECT to_tsvector('tst', 'indices');
 to_tsvector
-------------
 'index':1
(1 row)

mydb=# SELECT to_tsquery('tst', 'indices');
 to_tsquery
------------
 'index':*
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector;
            tsvector
--------------------------------
 'are' 'indexes' 'useful' 'very'
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector @@ to_tsquery('tst', 'indices');
 ?column?
----------
```

```
 t
(1 row)
```

12.6.4. Тезаурус

Тезаурус (или сокращённо TZ) содержит набор слов и информацию о связях слов и словосочетаний, то есть более широкие понятия (Broader Terms, BT), более узкие понятия (Narrow Terms, NT), предпочитаемые названия, исключаемые названия, связанные понятия и т. д.

В основном тезаурус заменяет исключаемые слова и словосочетания предпочитаемыми и может также сохранить исходные слова для индексации. Текущая реализация тезауруса в Postgres Pro представляет собой расширение словаря синонимов с поддержкой *фраз*. Конфигурация тезауруса определяется файлом следующего формата:

```
# это комментарий
образец слов(а) : индексируемые слова
другой образец слов(а) : другие индексируемые слова
...
```

Здесь двоеточие (`:`) служит разделителем между исходной фразой и её заменой.

Прежде чем проверять соответствие фраз, тезаурус нормализует файл конфигурации, используя *внутренний словарь* (который указывается в конфигурации словаря-тезауруса). Этот внутренний словарь для тезауруса может быть только одним. Если он не сможет распознать какое-либо слово, произойдёт ошибка. В этом случае необходимо либо исключить это слово, либо добавить его во внутренний словарь. Также можно добавить звёздочку (`*`) перед индексируемыми словами, чтобы они не проверялись по внутреннему словарю, но все слова-образцы *должны* быть известны внутреннему словарю.

Если входному фрагменту соответствуют несколько фраз в этом списке, тезаурус выберет самое длинное определение, а если таких окажется несколько, самое последнее из них.

Выделить во фразе какие-то стоп-слова нельзя; вместо этого можно вставить `?` в том месте, где может оказаться стоп-слово. Например, в предположении, что `a` и `the` — стоп-слова по внутреннему словарю:

```
? one ? two : swsw
```

соответствует входным строкам `a one the two` и `the one a two`, так что обе эти строки будут заменены на `swsw`.

Как и обычный словарь, тезаурус должен привязываться к лексемам определённых типов. Так как тезаурус может распознавать фразы, он должен запоминать своё состояние и взаимодействовать с анализатором. Учитывая свои привязки, он может либо обрабатывать следующий фрагмент, либо прекратить накопление фразы. Поэтому настройка тезаурусов в системе требует особого внимания. Например, если привязать тезаурус только к типу фрагментов `asciiword`, тогда определение в тезаурусе `one 7` не будет работать, так как этот тезаурус не связан с типом `uint`.

> **Внимание**
>
> Тезаурусы используются при индексации, поэтому при любом изменении параметров или содержимого тезауруса *необходима* переиндексация. Для большинства других типов словарей при небольших изменениях, таких как удаление и добавление стоп-слов, переиндексация не требуется.

12.6.4.1. Конфигурация тезауруса

Для создания нового словаря-тезауруса используется шаблон `thesaurus`. Например:

```
CREATE TEXT SEARCH DICTIONARY thesaurus_simple (
```

```
    TEMPLATE = thesaurus,
    DictFile = mythesaurus,
    Dictionary = pg_catalog.english_stem
);
```

Здесь:

- `thesaurus_simple` — имя нового словаря
- `mythesaurus` — базовое имя файла конфигурации тезауруса. (Полным путём к файлу будет `$SHAREDIR/tsearch_data/mythesaurus.ths`, где `$SHAREDIR` указывает на каталог общих данных PostgreSQL.)
- `pg_catalog.english_stem` — внутренний словарь (в данном случае это стеммер Snowball для английского) для нормализации тезауруса. Заметьте, что внутренний словарь имеет собственную конфигурацию (например, список стоп-слов), но здесь она не рассматривается.

Теперь тезаурус `thesaurus_simple` можно связать с желаемыми типами фрагментов в конфигурации, например так:

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_simple;
```

12.6.4.2. Пример тезауруса

Давайте рассмотрим простой астрономический тезаурус `thesaurus_astro`, содержащий несколько астрономических терминов:

```
supernovae stars : sn
crab nebulae : crab
```

Ниже мы создадим словарь и привяжем некоторые типы фрагментов к астрономическому тезаурусу и английскому стеммеру:

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (
    TEMPLATE = thesaurus,
    DictFile = thesaurus_astro,
    Dictionary = english_stem
);

ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_astro, english_stem;
```

Теперь можно проверить, как он работает. Функция `ts_lexize` не очень полезна для проверки тезауруса, так как она обрабатывает входную строку как один фрагмент. Вместо неё мы можем использовать функции `plainto_tsquery` и `to_tsvector`, которые разбивают входную строку на несколько фрагментов:

```
SELECT plainto_tsquery('supernova star');
 plainto_tsquery
-----------------
 'sn'

SELECT to_tsvector('supernova star');
 to_tsvector
-------------
 'sn':1
```

В принципе так же можно использовать `to_tsquery`, если заключить аргумент в кавычки:

```
SELECT to_tsquery(' ''supernova star''');
 to_tsquery
------------
 'sn'
```

Заметьте, что `supernova  star` совпадает с `supernovae  stars` в `thesaurus_astro`, так как мы подключили стеммер `english_stem` в определении тезауруса. Этот стеммер удалил конечные буквы `e` и `s`.

Чтобы проиндексировать исходную фразу вместе с заменой, её нужно просто добавить в правую часть соответствующего определения:

```
supernovae stars : sn supernovae stars

SELECT plainto_tsquery('supernova star');
       plainto_tsquery
-----------------------------
 'sn' & 'supernova' & 'star'
```

12.6.5. Словарь Ispell

Шаблон словарей Ispell поддерживает *морфологические словари*, которые могут сводить множество разных лингвистических форм слова к одной лексеме. Например, английский словарь Ispell может связать вместе все склонения и спряжения ключевого слова `bank`: `banking`, `banked`, `banks`, `banks'`,`bank's` и т. п.

Стандартный дистрибутив Postgres Pro не включает файлы конфигурации Ispell. Загрузить словари для множества языков можно со страницы *Ispell*. Кроме того, поддерживаются и другие современные форматы словарей: *MySpell* (OO < 2.0.1) и *Hunspell* (OO >= 2.0.2). Большой набор соответствующих словарей можно найти на странице *OpenOffice Wiki*.

Чтобы создать словарь Ispell, выполните следующие действия:

- загрузите файлы конфигурации словаря. Пакет с дополнительным словарём OpenOffice имеет расширение `.oxt`. Из него необходимо извлечь файлы `.aff` и `.dic`, и сменить их расширения на `.affix` и `.dict`, соответственно. Для некоторых файлов словарей также необходимо преобразовать символы в кодировку UTF-8 с помощью, например, таких команд (для норвежского языка):

  ```
  iconv -f ISO_8859-1 -t UTF-8 -o nn_no.affix nn_NO.aff
  iconv -f ISO_8859-1 -t UTF-8 -o nn_no.dict nn_NO.dic
  ```
- скопируйте файлы в каталог `$SHAREDIR/tsearch_data`
- загрузите эти файлы в Postgres Pro следующей командой:

  ```
  CREATE TEXT SEARCH DICTIONARY english_hunspell (
      TEMPLATE = ispell,
      DictFile = en_us,
      AffFile = en_us,
      Stopwords = english);
  ```

Здесь параметры `DictFile`, `AffFile` и `StopWords` определяют базовые имена файлов словаря, аффиксов и стоп-слов. Файл стоп-слов должен иметь тот же формат, что рассматривался выше в описании словаря `simple`. Формат других файлов здесь не рассматривается, но его можно узнать по вышеуказанным веб-адресам.

Словари Ispell обычно воспринимают ограниченный набор слов, так что за ними следует подключить другой, более общий словарь, например, Snowball, который принимает всё.

Файл `.affix` для Ispell имеет такую структуру:

```
prefixes
flag *A:
    .           >   RE      # As in enter > reenter
suffixes
flag T:
    E           >   ST      # As in late > latest
```

```
    [^AEIOU]Y   >   -Y,IEST # As in dirty > dirtiest
    [AEIOU]Y    >   EST     # As in gray > grayest
    [^EY]       >   EST     # As in small > smallest
```

А файл `.dict` — такую:

```
lapse/ADGRS
lard/DGRS
large/PRTY
lark/MRS
```

Формат файла `.dict` следующий:

```
basic_form/affix_class_name
```

В файле `.affix` каждый флаг аффиксов описывается в следующем формате:

```
условие > [-отсекаемые_буквы,] добавляемый_аффикс
```

Здесь условие записывается в формате, подобном формату регулярных выражений. В нём возможно описать группы `[...]` и `[^...]`. Например, запись `[AEIOU]Y` означает, что последняя буква слова — "`y`", а предпоследней может быть "`a`", "`e`", "`i`", "`o`" или "`u`". Запись `[^EY]` означает, что последняя буква не "`e`" и не "`y`".

Словари Ispell поддерживают разделение составных слов, что бывает полезно. Заметьте, что для этого в файле аффиксов нужно пометить специальным оператором `compoundwords controlled` слова, которые могут участвовать в составных образованиях:

```
compoundwords  controlled z
```

Вот как это работает для норвежского языка:

```
SELECT ts_lexize('norwegian_ispell',
  'overbuljongterningpakkmesterassistent');
   {over,buljong,terning,pakk,mester,assistent}
SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');
   {sjokoladefabrikk,sjokolade,fabrikk}
```

Формат MySpell представляет собой подмножество формата Hunspell. Файл `.affix` словаря Hunspell имеет следующую структуру:

```
PFX A Y 1
PFX A   0     re         .
SFX T N 4
SFX T   0     st         e
SFX T   y     iest       [^aeiou]y
SFX T   0     est        [aeiou]y
SFX T   0     est        [^ey]
```

Первая строка класса аффиксов — заголовок. Поля правил аффиксов указываются после заголовка:

- имя параметра (PFX или SFX)
- флаг (имя класса аффиксов)
- отсекаемые символы в начале (в префиксе) или в конце (в суффиксе) слова
- добавляемый аффикс
- условие в формате, подобном регулярным выражениям.

Файл `.dict` подобен файлу `.dict` словаря Ispell:

```
larder/M
lardy/RT
large/RSPMYT
```

```
largehearted
```

> **Примечание**
>
> Словарь MySpell не поддерживает составные слова. С другой стороны, Hunspell поддерживает множество операции с ними, но в настоящее время Postgres Pro использует только самые простые из этого множества.

12.6.6. Словарь Snowball

Шаблон словарей Snowball основан на проекте Мартина Потера, изобретателя популярного алгоритма стемминга для английского языка. Сейчас Snowball предлагает алгоритмы и для многих других языков (за подробностями обратитесь на *сайт Snowball*). Каждый алгоритм знает, как для данного языка свести распространённые словоформы к начальной форме. Для словаря Snowball задаётся обязательный параметр `language`, определяющий, какой именно стеммер использовать, и может задаваться параметр `stopword`, указывающий файл со списком исключаемых слов. (Стандартные списки стоп-слов Postgres Pro используется также в и проекте Snowball.) Например, встроенное определение выглядит так

```
CREATE TEXT SEARCH DICTIONARY english_stem (
    TEMPLATE = snowball,
    Language = english,
    StopWords = english
);
```

Формат файла стоп-слов не отличается от рассмотренного ранее.

Словарь Snowball распознаёт любые фрагменты, даже если он не может упростить слова, так что он должен быть самым последним в списке словарей. Помещать его перед другими словарями нет смысла, так как после него никакой фрагмент не будет передан следующему словарю.

12.7. Пример конфигурации

Конфигурация текстового поиска определяет всё, что необходимо для преобразования документа в формат `tsvector`: анализатор, который будет разбивать текст на фрагменты, и словари, которые будут преобразовывать фрагменты в лексемы. При каждом вызове `to_tsvector` или `to_tsquery` обязательно используется конфигурация текстового поиска. В конфигурации сервера есть параметр default_text_search_config, задающий имя конфигурации текстового поиска по умолчанию, которая будет использоваться, когда при вызове функций поиска соответствующий аргумент не определён. Этот параметр можно задать в `postgresql.conf` или установить в рамках отдельного сеанса с помощью команды `SET`.

В системе есть несколько встроенных конфигураций текстового поиска и вы можете легко дополнить их своими. Для удобства управления объектами текстового поиска в PostgreSQL реализованы соответствующие SQL-команды и специальные команды в psql, выводящие информацию об этих объектах (Раздел 12.10).

В качестве примера использования этих команд мы создадим конфигурацию `pg`, взяв за основу встроенную конфигурацию `english`:

```
CREATE TEXT SEARCH CONFIGURATION public.pg ( COPY = pg_catalog.english );
```

Мы будем использовать список синонимов, связанных с Postgres Pro, в файле `$SHAREDIR/tsearch_data/pg_dict.syn`. Этот файл содержит строки:

```
postgres    pg
pgsql       pg
postgresql  pg
```

Мы определим словарь синонимов следующим образом:

```
CREATE TEXT SEARCH DICTIONARY pg_dict (
    TEMPLATE = synonym,
    SYNONYMS = pg_dict
);
```

Затем мы зарегистрируем словарь Ispell `english_ispell`, у которого есть собственные файлы конфигурации:

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

Теперь мы можем настроить сопоставления для слов в конфигурации `pg`:

```
ALTER TEXT SEARCH CONFIGURATION pg
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                      word, hword, hword_part
    WITH pg_dict, english_ispell, english_stem;
```

Мы решили не индексировать и не учитывать при поиске некоторые типы фрагментов, которые не обрабатываются встроенной конфигурацией:

```
ALTER TEXT SEARCH CONFIGURATION pg
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

Теперь мы можем протестировать нашу конфигурацию:

```
SELECT * FROM ts_debug('public.pg', '
Postgres Pro, the highly scalable, SQL compliant, open source
object-relational database management system, is now undergoing
beta testing of the next version of our software.
');
```

И наконец мы выбираем в текущем сеансе эту конфигурацию, созданную в схеме `public`:

```
=> \dF
   List of text search configurations
 Schema  | Name | Description
---------+------+-------------
 public  | pg   |

SET default_text_search_config = 'public.pg';
SET

SHOW default_text_search_config;
 default_text_search_config
----------------------------
 public.pg
```

12.8. Тестирование и отладка текстового поиска

Поведение нестандартной конфигурации текстового поиска по мере её усложнения может стать непонятным. В этом разделе описаны функции, полезные для тестирования объектов текстового поиска. Вы можете тестировать конфигурацию как целиком, так и по частям, отлаживая анализаторы и словари по отдельности.

12.8.1. Тестирование конфигурации

Созданную конфигурацию текстового поиска можно легко протестировать с помощью функции `ts_debug`.

```
ts_debug([конфигурация regconfig,] документ text,
         OUT псевдоним text,
         OUT описание text,
         OUT фрагмент text,
         OUT словари regdictionary[],
         OUT словарь regdictionary,
         OUT лексемы text[])
         returns setof record
```

`ts_debug` выводит информацию обо всех фрагментах данного документа, которые были выданы анализатором и обработаны настроенными словарями. Она использует конфигурацию, указанную в аргументе *config*, или `default_text_search_config`, если этот аргумент опущен.

`ts_debug` возвращает по одной строке для каждого фрагмента, найденного в тексте анализатором. Эта строка содержит следующие столбцы:

- *синоним* `text` — краткое имя типа фрагмента
- *описание* `text` — описание типа фрагмента
- *фрагмент* `text` — текст фрагмента
- *словари* `regdictionary[]` — словари, назначенные в конфигурации для фрагментов такого типа
- *словарь* `regdictionary` — словарь, распознавший этот фрагмент, или `NULL`, если подходящего словаря не нашлось
- *лексемы* `text[]` — лексемы, выданные словарём, распознавшим фрагмент, или `NULL`, если подходящий словарь не нашёлся; может быть также пустым массивом (`{}`), если фрагмент распознан как стоп-слово

Простой пример:

```
SELECT * FROM ts_debug('english', 'a fat  cat sat on a mat - it ate a fat rats');
   alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | cat   | {english_stem} | english_stem | {cat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | sat   | {english_stem} | english_stem | {sat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | on    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | mat   | {english_stem} | english_stem | {mat}
 blank     | Space symbols   |       | {}             |              |
 blank     | Space symbols   | -     | {}             |              |
 asciiword | Word, all ASCII | it    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | ate   | {english_stem} | english_stem | {ate}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | rats  | {english_stem} | english_stem | {rat}
```

Для более полной демонстрации мы сначала создадим конфигурацию `public.english` и словарь Ispell для английского языка:

```
CREATE TEXT SEARCH CONFIGURATION public.english
  ( COPY = pg_catalog.english );

CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);

ALTER TEXT SEARCH CONFIGURATION public.english
   ALTER MAPPING FOR asciiword WITH english_ispell, english_stem;

SELECT * FROM ts_debug('public.english', 'The Brightest supernovaes');
   alias   |   description   |    token    |         dictionaries          |
 dictionary   |   lexemes
-----------+-----------------+-------------+-------------------------------
+----------------+-------------
 asciiword | Word, all ASCII | The         | {english_ispell,english_stem} |
 english_ispell | {}
 blank     | Space symbols   |             | {}                            |
     |
 asciiword | Word, all ASCII | Brightest   | {english_ispell,english_stem} |
 english_ispell | {bright}
 blank     | Space symbols   |             | {}                            |
     |
 asciiword | Word, all ASCII | supernovaes | {english_ispell,english_stem} |
 english_stem   | {supernova}
```

В этом примере слово `Brightest` было воспринято анализатором как фрагмент `ASCII  word` (синоним `asciiword`). Для этого типа фрагментов список словарей включает `english_ispell` и `english_stem`. Данное слово было распознано словарём `english_ispell`, который свёл его к `bright`. Слово `supernovaes` оказалось незнакомо словарю `english_ispell`, так что оно было передано следующему словарю, который его благополучно распознал (на самом деле `english_stem` — это стеммер Snowball, который распознаёт всё, поэтому он включён в список словарей последним).

Слово `The` было распознано словарём `english_ispell` как стоп-слово (см. Подраздел 12.6.1) и поэтому не будет индексироваться. Пробелы тоже отбрасываются, так как в данной конфигурации для них нет словарей.

Вы можете уменьшить ширину вывода, явно перечислив только те столбцы, которые вы хотите видеть:

```
SELECT alias, token, dictionary, lexemes
FROM ts_debug('public.english', 'The Brightest supernovaes');
   alias   |    token    |   dictionary   |   lexemes
-----------+-------------+----------------+-------------
 asciiword | The         | english_ispell | {}
 blank     |             |                |
 asciiword | Brightest   | english_ispell | {bright}
 blank     |             |                |
 asciiword | supernovaes | english_stem   | {supernova}
```

12.8.2. Тестирование анализатора

Следующие функции позволяют непосредственно протестировать анализатор текстового поиска.

```
ts_parse(имя_анализатора text, документ text,
         OUT код_фрагмента integer, OUT фрагмент text) returns setof record
ts_parse(oid_анализатора oid, документ text,
```

```
        OUT код_фрагмента integer, OUT фрагмент text) returns setof record
```

`ts_parse` разбирает данный документ и возвращает набор записей, по одной для каждого извлечённого фрагмента. Каждая запись содержит `код_фрагмента`, код назначенного типа фрагмента, и `фрагмент`, собственно текст фрагмента. Например:

```
SELECT * FROM ts_parse('default', '123 - a number');
 tokid | token
-------+--------
    22 | 123
    12 |
    12 | -
     1 | a
    12 |
     1 | number
```

```
ts_token_type(имя_анализатора text, OUT код_фрагмента integer,
              OUT псевдоним text, OUT описание text) returns setof record
ts_token_type(oid_анализатора oid, OUT код_фрагмента integer,
              OUT псевдоним text, OUT описание text) returns setof record
```

`ts_token_type` возвращает таблицу, описывающую все типы фрагментов, которые может распознать анализатор. Для каждого типа в этой таблице указывается целочисленный `tokid` (идентификатор), который анализатор использует для пометки фрагмента этого типа, `alias` (псевдоним), с которым этот тип фигурирует в командах конфигурации, и `description` (краткое описание). Например:

```
SELECT * FROM ts_token_type('default');
 tokid |      alias      |               description
-------+-----------------+------------------------------------------
     1 | asciiword       | Word, all ASCII
     2 | word            | Word, all letters
     3 | numword         | Word, letters and digits
     4 | email           | Email address
     5 | url             | URL
     6 | host            | Host
     7 | sfloat          | Scientific notation
     8 | version         | Version number
     9 | hword_numpart   | Hyphenated word part, letters and digits
    10 | hword_part      | Hyphenated word part, all letters
    11 | hword_asciipart | Hyphenated word part, all ASCII
    12 | blank           | Space symbols
    13 | tag             | XML tag
    14 | protocol        | Protocol head
    15 | numhword        | Hyphenated word, letters and digits
    16 | asciihword      | Hyphenated word, all ASCII
    17 | hword           | Hyphenated word, all letters
    18 | url_path        | URL path
    19 | file            | File or path name
    20 | float           | Decimal notation
    21 | int             | Signed integer
    22 | uint            | Unsigned integer
    23 | entity          | XML entity
```

12.8.3. Тестирование словаря

Для тестирования словаря предназначена функция `ts_lexize`.

```
ts_lexize(словарь regdictionary, фрагмент text) returns text[]
```

`ts_lexize` возвращает массив лексем, если входной *фрагмент* известен словарю, либо пустой массив, если этот фрагмент считается в словаре стоп-словом, либо `NULL`, если он не был распознан.

Примеры:

```
SELECT ts_lexize('english_stem', 'stars');
 ts_lexize
-----------
 {star}

SELECT ts_lexize('english_stem', 'a');
 ts_lexize
-----------
 {}
```

> **Примечание**
>
> Функция `ts_lexize` принимает одиночный *фрагмент*, а не просто текст. Вот пример возможного заблуждения:
>
> ```
> SELECT ts_lexize('thesaurus_astro', 'supernovae stars') is null;
>  ?column?
> ----------
>  t
> ```
>
> Хотя фраза `supernovae  stars` есть в тезаурусе `thesaurus_astro`, `ts_lexize` не работает, так как она не разбирает входной текст, а воспринимает его как один фрагмент. Поэтому для проверки тезаурусов следует использовать функции `plainto_tsquery` и `to_tsvector`, например:
>
> ```
> SELECT plainto_tsquery('supernovae stars');
>  plainto_tsquery
> -----------------
>  'sn'
> ```

12.9. Типы индексов, предпочитаемые для текстового поиска

Для ускорения полнотекстового поиска можно использовать индексы двух видов: GIN и GiST. Заметьте, что эти индексы не требуются для поиска, но если по какому-то столбцу поиск выполняется регулярно, обычно желательно создать индекс.

Чтобы создать такой индекс, выполните одну из следующих команд:

`CREATE INDEX` *имя* `ON` *таблица* `USING GIN (`*столбец*`);`

Создаёт индекс на базе GIN (Generalized Inverted Index, Обобщённый Инвертированный Индекс). *Столбец* должен иметь тип `tsvector`.

`CREATE INDEX` *имя* `ON` *таблица* `USING GIST (`*столбец*`);`

Создаёт индекс на базе GiST (Generalized Search Tree, Обобщённое дерево поиска). Здесь *столбец* может иметь тип `tsvector` или `tsquery`.

Более предпочтительными для текстового поиска являются индексы GIN. Будучи инвертированными индексами, они содержат записи для всех отдельных слов (лексем) с компактным списком мест их вхождений. При поиске нескольких слов можно найти первое, а затем воспользоваться индексом и исключить строки, в которых дополнительные слова отсутствуют. Индексы GIN хранят только слова (лексемы) из значений `tsvector`, и теряют

информацию об их весах. Таким образом для выполнения запроса с весами потребуется перепроверить строки в таблице.

Индекс GiST допускает *неточности*, то есть он допускает ложные попадания и поэтому их нужно исключать дополнительно, сверяя результат с фактическими данными таблицы. (Postgres Pro делает это автоматически.) Индексы GiST являются неточными, так как все документы в них представляются сигнатурой фиксированной длины. Эта сигнатура создаётся в результате представления присутствия каждого слова как одного бита в строке из n-бит, а затем логического объединения этих битовых строк. Если двум словам будет соответствовать одна битовая позиция, попадание оказывается ложным. Если для всех слов оказались установлены соответствующие биты (в случае фактического или ложного попадания), для проверки правильности предположения о совпадении слов необходимо прочитать строку таблицы.

Неточность индекса приводит к снижению производительности из-за дополнительных обращений к записям таблицы, для которых предположение о совпадении оказывается ложным. Так как произвольный доступ к таблице обычно не бывает быстрым, это ограничивает применимость индексов GiST. Вероятность ложных попаданий зависит от ряда факторов, например от количества уникальных слов, так что его рекомендуется сокращать, применяя словари.

Заметьте, что построение индекса GIN часто можно ускорить, увеличив maintenance_work_mem, тогда как время построения индекса GiST не зависит от этого параметра.

Правильно используя индексы GIN и GiST и разделяя большие коллекции документов на секции, можно реализовать очень быстрый поиск с возможностью обновления «на лету». Секционировать данные можно как на уровне базы, с использованием наследования таблиц, так и распределив документы по разным серверам и затем собирая результаты с помощью модуля dblink. Последний вариант возможен благодаря тому, что функции ранжирования используют только локальную информацию.

12.10. Поддержка psql

Информацию об объектах конфигурации текстового поиска можно получить в psql с помощью следующего набора команд:

```
\dF{d,p,t}[+] [ШАБЛОН]
```

Необязательный + в этих командах включает более подробный вывод.

В необязательном параметре *ШАБЛОН* может указываться имя объекта текстового поиска (возможно, дополненное схемой). Если *ШАБЛОН* не указан, выводится информация обо всех видимых объектах. *ШАБЛОН* может содержать регулярное выражение с *разными* масками для схемы и объекта. Это иллюстрируют следующие примеры:

```
=> \dF *fulltext*
       List of text search configurations
 Schema |  Name        | Description
--------+--------------+-------------
 public | fulltext_cfg |
```

```
=> \dF *.fulltext*
       List of text search configurations
 Schema   |  Name        | Description
----------+--------------------------
 fulltext | fulltext_cfg |
 public   | fulltext_cfg |
```

Возможны следующие команды:

```
\dF[+] [ШАБЛОН]
```

Список конфигураций текстового поиска (добавьте + для дополнительных сведений).

```
=> \dF russian
```

```
            List of text search configurations
   Schema   |  Name   |            Description
------------+---------+-----------------------------------
 pg_catalog | russian | configuration for russian language

=> \dF+ russian
Text search configuration "pg_catalog.russian"
Parser: "pg_catalog.default"
      Token      | Dictionaries
-----------------+--------------
 asciihword      | english_stem
 asciiword       | english_stem
 email           | simple
 file            | simple
 float           | simple
 host            | simple
 hword           | russian_stem
 hword_asciipart | english_stem
 hword_numpart   | simple
 hword_part      | russian_stem
 int             | simple
 numhword        | simple
 numword         | simple
 sfloat          | simple
 uint            | simple
 url             | simple
 url_path        | simple
 version         | simple
 word            | russian_stem
```

`\dFd[+] [ШАБЛОН]`

Список словарей текстового поиска (добавьте + для дополнительных сведений).

```
=> \dFd
                            List of text search dictionaries
  Schema   |     Name       |              Description
-----------+----------------+-----------------------------------------
pg_catalog | danish_stem    | snowball stemmer for danish language
pg_catalog | dutch_stem     | snowball stemmer for dutch language
pg_catalog | english_stem   | snowball stemmer for english language
pg_catalog | finnish_stem   | snowball stemmer for finnish language
pg_catalog | french_stem    | snowball stemmer for french language
pg_catalog | german_stem    | snowball stemmer for german language
pg_catalog | hungarian_stem | snowball stemmer for hungarian language
pg_catalog | italian_stem   | snowball stemmer for italian language
pg_catalog | norwegian_stem | snowball stemmer for norwegian language
pg_catalog | portuguese_stem| snowball stemmer for portuguese language
pg_catalog | romanian_stem  | snowball stemmer for romanian language
pg_catalog | russian_stem   | snowball stemmer for russian language
pg_catalog | simple         | simple dictionary: just lower case and ...
pg_catalog | spanish_stem   | snowball stemmer for spanish language
pg_catalog | swedish_stem   | snowball stemmer for swedish language
pg_catalog | turkish_stem   | snowball stemmer for turkish language
```

`\dFp[+] [ШАБЛОН]`

Список анализаторов текстового поиска (добавьте + для дополнительных сведений).

```
=> \dFp
```

```
        List of text search parsers
   Schema   |  Name   |     Description
------------+---------+---------------------
 pg_catalog | default | default word parser
=> \dFp+
    Text search parser "pg_catalog.default"
     Method      |    Function    | Description
-----------------+----------------+-------------
 Start parse     | prsd_start     |
 Get next token  | prsd_nexttoken |
 End parse       | prsd_end       |
 Get headline    | prsd_headline  |
 Get token types | prsd_lextype   |

        Token types for parser "pg_catalog.default"
   Token name    |               Description
-----------------+------------------------------------------
 asciihword      | Hyphenated word, all ASCII
 asciiword       | Word, all ASCII
 blank           | Space symbols
 email           | Email address
 entity          | XML entity
 file            | File or path name
 float           | Decimal notation
 host            | Host
 hword           | Hyphenated word, all letters
 hword_asciipart | Hyphenated word part, all ASCII
 hword_numpart   | Hyphenated word part, letters and digits
 hword_part      | Hyphenated word part, all letters
 int             | Signed integer
 numhword        | Hyphenated word, letters and digits
 numword         | Word, letters and digits
 protocol        | Protocol head
 sfloat          | Scientific notation
 tag             | XML tag
 uint            | Unsigned integer
 url             | URL
 url_path        | URL path
 version         | Version number
 word            | Word, all letters
(23 rows)
```

`\dFt[+] [ШАБЛОН]`

Список шаблонов текстового поиска (добавьте + для дополнительных сведений).

```
=> \dFt
                            List of text search templates
  Schema  |  Name   |                        Description
----------+---------+-----------------------------------------------------
pg_catalog|ispell   |ispell dictionary
pg_catalog|simple   |simple dictionary: just lower case and check for ...
pg_catalog|snowball |snowball stemmer
pg_catalog|synonym  |synonym dictionary: replace word by its synonym
pg_catalog|thesaurus|thesaurus dictionary: phrase by phrase substitution
```

12.11. Ограничения

Текущая реализация текстового поиска в Postgres Pro имеет следующие ограничения:

- Длина лексемы не может превышать 2 килобайта
- Длина значения `tsvector` (лексемы и их позиции) не может превышать 1 мегабайт
- Число лексем должно быть меньше 2^{64}
- Значения позиций в `tsvector` должны быть от 0 до 16383
- Расстояние в операторе `<N>` (ПРЕДШЕСТВУЕТ) типа `tsquery` не может быть больше 16384
- Не больше 256 позиций для одной лексемы
- Число узлов (лексемы + операторы) в значении `tsquery` должно быть меньше 32768

Для сравнения, документация PostgreSQL 8.1 содержала 335 420 слов, из них 10 441 уникальных, а наиболее часто употребляющееся в ней слово «postgresql» встречается 6 127 раз в 655 документах.

Другой пример — архивы списков рассылки PostgreSQL содержали 910 989 уникальных слов в 57 491 343 лексемах в 461 020 сообщениях.

Глава 13. Управление конкурентным доступом

В этой главе описывается поведение СУБД Postgres Pro в ситуациях, когда два или более сеансов пытаются одновременно обратиться к одним и тем же данным. В таких ситуациях важно, чтобы все сеансы могли эффективно работать с данными, и при этом сохранялась целостность данных. Обсуждаемые в этой главе темы заслуживают внимания всех разработчиков баз данных.

13.1. Введение

Postgres Pro предоставляет разработчикам богатый набор средств для управления конкурентным доступом к данным. Внутри он поддерживает целостность данных, реализуя модель MVCC (Multiversion Concurrency Control, Многоверсионное управление конкурентным доступом). Это означает, что каждый SQL-оператор видит снимок данных (*версию базы данных*) на определённый момент времени, вне зависимости от текущего состояния данных. Это защищает операторы от несогласованности данных, возможной, если другие конкурирующие транзакции внесут изменения в те же строки данных, и обеспечивает тем самым *изоляцию транзакций* для каждого сеанса баз данных. MVCC, отходя от методик блокирования, принятых в традиционных СУБД, снижает уровень конфликтов блокировок и таким образом обеспечивает более высокую производительность в многопользовательской среде.

Основное преимущество использования модели MVCC по сравнению с блокированием заключается в том, что блокировки MVCC, полученные для чтения данных, не конфликтуют с блокировками, полученными для записи, и поэтому чтение никогда не мешает записи, а запись чтению. Postgres Pro гарантирует это даже для самого строгого уровня изоляции транзакций, используя инновационный уровень изоляции SSI (*Serializable Snapshot Isolation*, Сериализуемая изоляция снимков).

Для приложений, которым в принципе не нужна полная изоляция транзакций и которые предпочитают явно определять точки конфликтов, в Postgres Pro также есть средства блокировки на уровне таблиц и строк. Однако при правильном использовании MVCC обычно обеспечивает лучшую производительность, чем блокировки. Кроме этого, приложения могут использовать рекомендательные блокировки, не привязанные к какой-либо одной транзакции.

13.2. Изоляция транзакций

Стандарт SQL определяет четыре уровня изоляции транзакций. Наиболее строгий из них — сериализуемый, определяется одним абзацем, говорящем, что при параллельном выполнении несколько сериализуемых транзакций должны гарантированно выдавать такой же результат, как если бы они запускались по очереди в некотором порядке. Остальные три уровня определяются через описания особых явлений, которые возможны при взаимодействии параллельных транзакций, но не допускаются на определённом уровне. Как отмечается в стандарте, из определения сериализуемого уровня вытекает, что на этом уровне ни одно из этих явлений не возможно. (В самом деле — если эффект транзакций должен быть тем же, что и при их выполнении по очереди, как можно было бы увидеть особые явления, связанные с другими транзакциями?)

Стандарт описывает следующие особые условия, недопустимые для различных уровней изоляции:

«грязное» чтение

Транзакция читает данные, записанные параллельной незавершённой транзакцией.

неповторяемое чтение

Транзакция повторно читает те же данные, что и раньше, и обнаруживает, что они были изменены другой транзакцией (которая завершилась после первого чтения).

фантомное чтение

Транзакция повторно выполняет запрос, возвращающий набор строк для некоторого условия, и обнаруживает, что набор строк, удовлетворяющих условию, изменился из-за транзакции, завершившейся за это время.

аномалия сериализации

Результат успешной фиксации группы транзакций оказывается несогласованным при всевозможных вариантах исполнения этих транзакций по очереди.

Уровни изоляции транзакций, описанные в стандарте SQL и реализованные в Postgres Pro, описываются в Таблице 13.1.

Таблица 13.1. Уровни изоляции транзакций

Уровень изоляции	«Грязное» чтение	Неповторяемое чтение	Фантомное чтение	Аномалия сериализации
Read uncommited (Чтение незафиксированных данных)	Допускается, но не в PG	Возможно	Возможно	Возможно
Read committed (Чтение зафиксированных данных)	Невозможно	Возможно	Возможно	Возможно
Repeatable read (Повторяемое чтение)	Невозможно	Невозможно	Допускается, но не в PG	Возможно
Serializable (Сериализуемость)	Невозможно	Невозможно	Невозможно	Невозможно

В Postgres Pro вы можете запросить любой из четырёх уровней изоляции транзакций, однако внутри реализованы только три различных уровня, то есть режим Read Uncommitted в Postgres Pro действует как Read Committed. Причина этого в том, что только так можно сопоставить стандартные уровни изоляции с реализованной в Postgres Pro архитектурой многоверсионного управления конкурентным доступом.

В этой таблице также показано, что реализация Repeatable Read в Postgres Pro не допускает фантомного чтения. Это соответствует стандарту SQL, поскольку он устанавливает, каких аномалий *не* должно быть на определённых уровнях изоляции, но допускает и более строгие ограничения. Поведение имеющихся уровней изоляции подробно описывается в следующих подразделах.

Для выбора нужного уровня изоляции транзакций используется команда SET TRANSACTION.

Важно

Поведение некоторых функций и типов данных Postgres Pro в транзакциях подчиняется особым правилам. В частности, изменения последовательностей (и следовательно, счётчика в столбце, объявленному как `serial`) немедленно видны во всех остальных транзакциях и не откатываются назад, если выполнившая их транзакция прерывается. См. Раздел 9.16 и Подраздел 8.1.4.

13.2.1. Уровень изоляции Read Committed

Read Committed — уровень изоляции транзакции, выбираемый в Postgres Pro по умолчанию. В транзакции, работающей на этом уровне, запрос `SELECT` (без предложения `FOR UPDATE/`

SHARE) видит только те данные, которые были зафиксированы до начала запроса; он никогда не увидит незафиксированных данных или изменений, внесённых в процессе выполнения запроса параллельными транзакциями. По сути запрос SELECT видит снимок базы данных в момент начала выполнения запроса. Однако SELECT видит результаты изменений, внесённых ранее в этой же транзакции, даже если они ещё не зафиксированы. Также заметьте, что два последовательных оператора SELECT могут видеть разные данные даже в рамках одной транзакции, если какие-то другие транзакции зафиксируют изменения после запуска первого SELECT, но до запуска второго.

Команды UPDATE, DELETE, SELECT FOR UPDATE и SELECT FOR SHARE ведут себя подобно SELECT при поиске целевых строк: они найдут только те целевые строки, которые были зафиксированы на момент начала команды. Однако к моменту, когда они будут найдены, эти целевые строки могут быть уже изменены (а также удалены или заблокированы) другой параллельной транзакцией. В этом случае запланированное изменение будет отложено до фиксирования или отката первой изменяющей данные транзакции (если она ещё выполняется). Если первая изменяющая транзакция откатывается, её результат отбрасывается и вторая изменяющая транзакция может продолжить изменение изначально полученной строки. Если первая транзакция зафиксировалась, но в результате удалила эту строку, вторая будет игнорировать её, а в противном случае попытается выполнить свою операцию с изменённой версией строки. Условие поиска в команде (предложение WHERE) вычисляется повторно для выяснения, соответствует ли по-прежнему этому условию изменённая версия строки. Если да, вторая изменяющая транзакция продолжают свою работу с изменённой версией строки. Применительно к командам SELECT FOR UPDATE и SELECT FOR SHARE это означает, что изменённая версия строки блокируется и возвращается клиенту.

Похожим образом ведёт себя INSERT с предложением ON CONFLICT DO UPDATE. В режиме Read Committed каждая строка, предлагаемая для добавления, будет либо вставлена, либо изменена. Если не возникнет несвязанных ошибок, гарантируется один из этих двух исходов. Если конфликт будет вызван другой транзакцией, результат которой ещё не видим для INSERT, предложение UPDATE подействует на эту строку, даже несмотря на то, что эта команда обычным образом может не видеть *никакую* версию этой строки.

При выполнении INSERT с предложением ON CONFLICT DO NOTHING строка может не добавиться в результате действия другой транзакции, эффект которой не виден в снимке команды INSERT. Это опять же имеет место только в режиме Read Committed.

Вследствие описанных выше правил, изменяющая команда может увидеть несогласованное состояние: она может видеть результаты параллельных команд, изменяющих те же строки, что пытается изменить она, но при этом она не видит результаты этих команд в других строках таблиц. Из-за этого поведения уровень Read Committed не подходит для команд со сложными условиями поиска; однако он вполне пригоден для простых случаев. Например, рассмотрим изменение баланса счёта в таких транзакциях:

```
BEGIN;
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 12345;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 7534;
COMMIT;
```

Если две такие транзакции пытаются параллельно изменить баланс счёта 12345, мы, естественно, хотим, чтобы вторая транзакция работала с изменённой версией строки счёта. Так как каждая команда влияет только на определённую строку, если она будет видеть изменённую версию строки, это не приведёт к проблемам несогласованности.

В более сложных ситуациях уровень Read Committed может приводить к нежелательным результатам. Например, рассмотрим команду DELETE, работающую со строками, которые параллельно добавляет и удаляет из множества, определённого её условием, другая команда. Например, предположим, что website — таблица из двух строк, в которых website.hits равны 9 и 10:

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- выполняется параллельно:  DELETE FROM website WHERE hits = 10;
```

```
COMMIT;
```

Команда `DELETE` не сделает ничего, даже несмотря на то, что строка с `website.hits = 10` была в таблице и до, и после выполнения `UPDATE`. Это происходит потому, что строка со значением `9` до изменения пропускается, а когда команда `UPDATE` завершается и `DELETE` получает освободившуюся блокировку, строка с `10` теперь содержит `11`, а это значение уже не соответствует условию.

Так как в режиме Read Committed каждая команда начинается с нового снимка состояния, который включает результаты всех транзакций, зафиксированных к этому моменту, последующие команды в одной транзакции будут в любом случае видеть эффекты всех параллельных зафиксированных транзакций. Вопрос здесь состоит в том, видит ли *одна* команда абсолютно согласованное состояние базы данных.

Частичная изоляция транзакций, обеспечиваемая в режиме Read Committed, приемлема для множества приложений. Этот режим быстр и прост в использовании, однако он подходит не для всех случаев. Приложениям, выполняющим сложные запросы и изменения, могут потребоваться более строго согласованное представление данных, чем то, что даёт Read Committed.

13.2.2. Уровень изоляции Repeatable Read

В режиме *Repeatable Read* видны только те данные, которые были зафиксированы до начала транзакции, но не видны незафиксированные данные и изменения, произведённые другими транзакциями в процессе выполнения данной транзакции. (Однако запрос будет видеть эффекты предыдущих изменений в своей транзакции, несмотря на то, что они не зафиксированы.) Это самое строгое требование, которое стандарт SQL вводит для этого уровня изоляции, и при его выполнении предотвращаются все явления, описанные в Таблице 13.1, за исключением аномалий сериализации. Как было сказано выше, это не противоречит стандарту, так как он определяет только *минимальную* защиту, которая должна обеспечиваться на каждом уровне изоляции.

Этот уровень отличается от Read Committed тем, что запрос в транзакции данного уровня видит снимок данных на момент начала первого оператора в *транзакции* (не считая команд управления транзакциями), а не начала текущего оператора. Таким образом, последовательные команды `SELECT` в *одной* транзакции видят одни и те же данные; они не видят изменений, внесённых и зафиксированных другими транзакциями после начала их текущей транзакции.

Приложения, использующие этот уровень, должны быть готовы повторить транзакции в случае сбоев сериализации.

Команды `UPDATE`, `DELETE`, `SELECT FOR UPDATE` и `SELECT FOR SHARE` ведут себя подобно `SELECT` при поиске целевых строк: они найдут только те целевые строки, которые были зафиксированы на момент начала транзакции. Однако к моменту, когда они будут найдены, эти целевые строки могут быть уже изменены (а также удалены или заблокированы) другой параллельной транзакцией. В этом случае транзакция в режиме Repeatable Read будет ожидать фиксирования или отката первой изменяющей данные транзакции (если она ещё выполняется). Если первая изменяющая транзакция откатывается, её результат отбрасывается и текущая транзакция может продолжить изменение изначально полученной строки. Если же первая транзакция зафиксировалась и в результате изменила или удалила эту строку, а не просто заблокировала её, произойдёт откат текущей транзакции с сообщением

```
ОШИБКА: не удалось сериализовать доступ из-за параллельного изменения
```

так как транзакция уровня Repeatable Read не может изменять или блокировать строки, изменённые другими транзакциями с момента её начала.

Когда приложение получает это сообщение об ошибке, оно должна прервать текущую транзакцию и попытаться повторить её с самого начала. Во второй раз транзакция увидит внесённое до этого изменение как часть начального снимка базы данных, так что новая версия строки вполне может использоваться в качестве отправной точки для изменения в повторной транзакции.

Заметьте, что потребность в повторении транзакции может возникнуть, только если эта транзакция изменяет данные; в транзакциях, которые только читают данные, конфликтов сериализации не бывает.

Режим Repeatable Read строго гарантирует, что каждая транзакция видит полностью стабильное представление базы данных. Однако это представление не обязательно будет согласовано с некоторым последовательным выполнением транзакций одного уровня. Например, даже транзакция, которая только читает данные, в этом режиме может видеть строку, показывающую, что некоторое задание завершено, но *не* видеть одну из строк логических частей задания, так как эта транзакция может прочитать более раннюю версию строки задания, чем ту, для которой параллельно добавлялась очередная логическая часть. Строго исполнить бизнес-правила в транзакциях, работающих на этом уровне изоляции, скорее всего не удастся без явных блокировок конфликтующих транзакций.

Для реализации уровня изоляции Repeatable Read применяется подход, который называется в академической литературе по базам данных и в других СУБД *Изоляция снимков* (Snapshot Isolation). По сравнению с системами, использующими традиционный метод блокировок, затрудняющий параллельное выполнение, при этом подходе наблюдается другое поведение и другая производительность. В некоторых СУБД могут существовать даже два отдельных уровня Repeatable Read и Snapshot Isolation с различным поведением. Допускаемые особые условия, представляющие отличия двух этих подходов, не были формализованы разработчиками теории БД до развития стандарта SQL и их рассмотрение выходит за рамки данного руководства. В полном объёме эта тема освещается в berenson95.

> **Примечание**
>
> До версии 9.1 в PostgreSQL при запросе режима Serializable поведение системы в точности соответствовало вышеописанному. Таким образом, чтобы сейчас получить старое поведение Serializable, нужно запрашивать режим Repeatable Read.

13.2.3. Уровень изоляции Serializable

Уровень *Serializable* обеспечивает самую строгую изоляцию транзакций. На этом уровне моделируется последовательное выполнение всех зафиксированных транзакций, как если бы транзакции выполнялись одна за другой, последовательно, а не параллельно. Однако, как и на уровне Repeatable Read, на этом уровне приложения должны быть готовы повторять транзакции из-за сбоев сериализации. Фактически этот режим изоляции работает так же, как и Repeatable Read, только он дополнительно отслеживает условия, при которых результат параллельно выполняемых сериализуемых транзакций может не согласовываться с результатом этих же транзакций, выполняемых по очереди. Это отслеживание не привносит дополнительных препятствий для выполнения, кроме тех, что присущи режиму Repeatable Read, но тем не менее создаёт некоторую добавочную нагрузку, а при выявлении исключительных условий регистрируется *аномалия сериализации* и происходит *сбой сериализации*.

Например, рассмотрим таблицу `mytab`, изначально содержащую:

```
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

Предположим, что сериализуемая транзакция A вычисляет:

```
SELECT SUM(value) FROM mytab WHERE class = 1;
```

а затем вставляет результат (30) в поле `value` в новую строку со значением `class = 2`. В это же время сериализуемая транзакция B вычисляет:

```
SELECT SUM(value) FROM mytab WHERE class = 2;
```

получает результат 300 и вставляет его в новую строку со значением `class = 1`. Затем обе транзакции пытаются зафиксироваться. Если бы одна из этих транзакций работала в режиме Repeatable Read, зафиксироваться могли бы обе; но так как полученный результат не

соответствовал бы последовательному порядку, в режиме Serializable будет зафиксирована только одна транзакция, а вторая закончится откатом с сообщением:

```
ОШИБКА: не удалось сериализовать доступ из-за зависимостей чтения/записи между
  транзакциями
```

Это объясняется тем, что при выполнении A перед B транзакция B вычислила бы сумму 330, а не 300, а при выполнении в обратном порядке A вычислила бы другую сумму.

Рассчитывая, что сериализуемые транзакции предотвратят аномалии, важно понимать, что любые данные, полученные из постоянной таблицы пользователя, не должны считаться действительными, пока транзакция, прочитавшая их, не будет успешно зафиксирована. Это верно даже для транзакций, не модифицирующих данные, за исключением случая, когда данные считываются в *откладываемой* транзакции такого типа. В этом случае данные могут считаться действительными, так как такая транзакция ждёт, пока не сможет получить снимок, гарантированно предотвращающий подобные проблемы. Во всех остальных случаях приложения не должны полагаться на результаты чтения данных в транзакции, которая не была зафиксирована; в случае ошибки и отката приложения должны повторять транзакцию, пока она не будет завершена успешно.

Для полной гарантии сериализуемости в Postgres Pro применяются *предикатные блокировки*, то есть блокировки, позволяющие определить, когда запись могла бы повлиять на результат предыдущего чтения параллельной транзакции, если бы эта запись выполнялась сначала. В Postgres Pro эти блокировки не приводят к фактическим блокировкам данных и, следовательно, никоим образом *не* могут повлечь взаимоблокировки транзакций. Они помогают выявить и отметить зависимости между параллельными транзакциями уровня Serializable, которые в определённых сочетаниях могут приводить к аномалиям сериализации. Транзакции Read Committed или Repeatable Read для обеспечения целостности данных, напротив, должны либо блокировать таблицы целиком, что помешает пользователям обращаться к этим таблицам, либо применять `SELECT FOR UPDATE` или `SELECT FOR SHARE`, что не только заблокирует другие транзакции, но и создаст дополнительную нагрузку на диск.

Предикатные блокировки в Postgres Pro, как и в большинстве других СУБД, устанавливаются для данных, фактически используемых в транзакции. Они отображаются в системном представлении `pg_locks` со значением `mode` равным `SIReadLock`. Какие именно блокировки будут затребованы при выполнении запроса, зависит от плана запроса, при этом детализированные блокировки (например, блокировки строк) могут объединяться в более общие (например, в блокировки страниц) в процессе транзакции для экономии памяти, расходуемой для отслеживания блокировок. Транзакция `READ ONLY` может даже освободить свои блокировки SIRead до завершения, если обнаруживается, что конфликты, которые могли бы привести к аномалии сериализации, исключены. На самом деле для транзакций `READ ONLY` этот факт чаще всего устанавливается в самом начале, так что они обходятся без предикатных блокировок. Если же вы явно запросите транзакцию `SERIALIZABLE READ ONLY DEFERRABLE`, она будет заблокирована до тех пор, пока не сможет установить этот факт. (Это *единственный* случай, когда транзакции уровня Serializable блокируются, а транзакции Repeatable Read — нет.) С другой стороны, блокировки SIRead часто должны сохраняться и после фиксирования транзакции, пока не будут завершены другие, наложившиеся на неё транзакции.

При правильном использовании сериализуемые транзакции могут значительно упростить разработку приложений. Гарантия того, что любое сочетание успешно зафиксированных параллельных сериализуемых транзакций даст тот же результат, что и последовательность этих транзакций, выполненных по очереди, означает, что если вы уверены, что единственная транзакция определённого содержания работает правильно, когда она запускается отдельно, вы можете быть уверены, что она будет работать так же правильно в любом сочетании сериализуемых транзакций, вне зависимости от того, что они делают, либо же она не будет зафиксирована успешно. При этом важно, чтобы в среде, где применяется этот подход, была реализована общая обработка сбоев сериализации (которые можно определить по значению SQLSTATE '40001'), так как заведомо определить, какие именно транзакции могут стать жертвами зависимостей чтения/записи и не будут зафиксированы для предотвращения аномалий сериализации, обычно очень сложно. Отслеживание зависимостей чтения-записи неизбежно создаёт дополнительную

нагрузку, как и перезапуск транзакций, не зафиксированных из-за сбоев сериализации, но если на другую чашу весов положить нагрузку и блокирование, связанные с применением явных блокировок и `SELECT FOR UPDATE` или `SELECT FOR SHARE`, использовать сериализуемые транзакции в ряде случаев окажется выгоднее.

Тогда как уровень изоляции транзакций Serializable в Postgres Pro позволяет фиксировать параллельные транзакции, только если есть уверенность, что тот же результат будет получен при последовательном их выполнении, он не всегда предотвращает ошибки, которые не возникли бы при действительно последовательном выполнении. В частности, можно столкнуться с нарушениями ограничений уникальности, вызванными наложением сериализуемых транзакций, даже после явной проверки отсутствия ключа перед добавлением его. Этого можно избежать, если *все* сериализуемые транзакции, добавляющие потенциально конфликтующие ключи, будут предварительно явно проверять, можно ли вставить ключ. Например, приложение, добавляющее новый ключ, может запрашивать его у пользователя и затем проверять, существует ли он, сначала пытаясь найти его, либо генерировать новый ключ, выбирая максимальное существующее значение и увеличивая его на один. Если некоторые сериализуемые транзакции добавляют новые ключи сразу, не следуя этому протоколу, возможны нарушения ограничений уникальности, даже когда они не наблюдались бы при последовательном выполнении этих транзакций.

Применяя сериализуемые транзакции для управления конкурентным доступом, примите к сведению следующие рекомендации:

- Объявляйте транзакции как `READ ONLY`, если это отражает их суть.
- Управляйте числом активных подключений, при необходимости используя пул соединений. Это всегда полезно для увеличения производительности, но особенно важно это в загруженной системе с сериализуемыми транзакциями.
- Заключайте в одну транзакцию не больше команд, чем необходимо для обеспечения целостности.
- Не оставляйте соединения «простаивающими в транзакции» дольше, чем необходимо. Для автоматического отключения затянувшихся транзакций можно применить параметр конфигурации idle_in_transaction_session_timeout.
- Исключите явные блокировки, `SELECT FOR UPDATE` и `SELECT FOR SHARE` там, где они не нужны благодаря защите, автоматически предоставляемой сериализуемыми транзакциями.
- Когда система вынуждена объединять предикатные блокировки уровня страницы в одну предикатную блокировку уровня таблицы из-за нехватки памяти, может возрасти частота сбоев сериализации. Избежать этого можно, увеличив параметр max_pred_locks_per_transaction, max_pred_locks_per_relation и/или max_pred_locks_per_page.
- Последовательное сканирование всегда влечёт за собой предикатную блокировку на уровне таблицы. Это приводит к увеличению сбоев сериализации. В таких ситуациях бывает полезно склонить систему к использованию индексов, уменьшая random_page_cost и/или увеличивая cpu_tuple_cost. Однако тут важно сопоставить выигрыш от уменьшения числа откатов и перезапусков транзакций с проигрышем от возможного менее эффективного выполнения запросов.

Для реализации уровня изоляции Serializable применяется подход, который называется в академической литературе по базам данных *Изоляция снимков* (Snapshot Isolation), с дополнительными проверками на предмет аномалий сериализации. По сравнению с другими системами, использующими традиционный метод блокировок, при этом подходе наблюдается другое поведение и другая производительность. Подробнее это освещается в ports12.

13.3. Явные блокировки

Для управления параллельным доступом к данным в таблицах Postgres Pro предоставляет несколько режимов явных блокировок. Эти режимы могут применяться для блокировки данных со стороны приложения в ситуациях, когда MVCC не даёт желаемый результат. Кроме того, большинство команд Postgres Pro автоматически получают блокировки соответствующих режимов, защищающие от удаления или изменения задействованных таблиц, несовместимого

с характером выполняемой команды. (Например, TRUNCATE не может безопасно выполняться одновременно с другими операциями с этой таблицей, так что во избежание конфликта эта команда получает блокировку ACCESS EXCLUSIVE для данной таблицы.)

Список текущих активных блокировок на сервере можно получить, прочитав системное представление pg_locks. За дополнительными сведениями о наблюдении за состоянием менеджера блокировок обратитесь к Главе 27.

13.3.1. Блокировки на уровне таблицы

В приведённом ниже списке перечислены имеющиеся режимы блокировок и контексты, где их автоматически применяет Postgres Pro. Вы можете также явно запросить любую из этих блокировок с помощью команды LOCK. Помните, что все эти режимы работают на уровне таблицы, даже если имя режима содержит слово «row»; такие имена сложились исторически. В некоторой степени эти имена отражают типичное применение каждого режима блокировки, но смысл у всех один. Единственное, что действительно отличает один режим блокировки от другого, это набор режимов, с которыми конфликтует каждый из них (см. Таблицу 13.2). Две транзакции не могут одновременно владеть блокировками конфликтующих режимов для одной и той же таблицы. (Однако учтите, что транзакция никогда не конфликтует с собой. Например, она может запросить блокировку ACCESS EXCLUSIVE, а затем ACCESS SHARE для той же таблицы.) При этом разные транзакции свободно могут одновременно владеть блокировками неконфликтующих режимов. Заметьте, что некоторые режимы блокировки конфликтуют сами с собой (например, блокировкой ACCESS EXCLUSIVE в один момент времени может владеть только одна транзакция), а некоторые — нет (например, блокировку ACCESS SHARE могут получить сразу несколько транзакций).

Режимы блокировок на уровне таблицы

ACCESS SHARE (AccessShareLock)

Конфликтует только с режимом блокировки ACCESS EXCLUSIVE.

Команда SELECT получает такую блокировку для таблиц, на которые она ссылается. Вообще говоря, блокировку в этом режиме получает любой запрос, который только *читает* таблицу, но не меняет её данные.

ROW SHARE (RowShareLock)

Конфликтует с режимами блокировки EXCLUSIVE и ACCESS EXCLUSIVE.

Команды SELECT FOR UPDATE и SELECT FOR SHARE получают такую блокировку для своих целевых таблиц (помимо блокировок ACCESS SHARE для любых таблиц, которые используется в этих запросах, но не в предложении FOR UPDATE/FOR SHARE).

ROW EXCLUSIVE (RowExclusiveLock)

Конфликтует с режимами блокировки SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE и ACCESS EXCLUSIVE.

Команды UPDATE, DELETE и INSERT получают такую блокировку для целевой таблицы (в дополнение к блокировкам ACCESS SHARE для всех других задействованных таблиц). Вообще говоря, блокировку в этом режиме получает любая команда, которая *изменяет данные* в таблице.

SHARE UPDATE EXCLUSIVE (ShareUpdateExclusiveLock)

Конфликтует с режимами блокировки SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE и ACCESS EXCLUSIVE. Этот режим защищает таблицу от параллельного изменения схемы и запуска процесса VACUUM.

Запрашивается командами VACUUM (без FULL), ANALYZE, CREATE INDEX CONCURRENTLY, CREATE STATISTICS, COMMENT ON, а также ALTER TABLE VALIDATE и другими видами ALTER TABLE (за подробностями обратитесь к ALTER TABLE).

SHARE (ShareLock)

Конфликтует с режимами блокировки ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE и ACCESS EXCLUSIVE. Этот режим защищает таблицу от параллельного изменения данных.

Запрашивается командой CREATE INDEX (без параметра CONCURRENTLY).

SHARE ROW EXCLUSIVE (ShareRowExclusiveLock)

Конфликтует с режимами блокировки ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE и ACCESS EXCLUSIVE. Этот режим защищает таблицу от параллельных изменений данных и при этом он является самоисключающим, так что такую блокировку может получить только один сеанс.

Запрашивается командой CREATE COLLATION, CREATE TRIGGER и многими формами ALTER TABLE (см. ALTER TABLE).

EXCLUSIVE (ExclusiveLock)

Конфликтует с режимами блокировки ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE и ACCESS EXCLUSIVE. Этот режим совместим только с блокировкой ACCESS SHARE, то есть параллельно с транзакцией, получившей блокировку в этом режиме, допускается только чтение таблицы.

Запрашивается командой REFRESH MATERIALIZED VIEW CONCURRENTLY.

ACCESS EXCLUSIVE (AccessExclusiveLock)

Конфликтует с режимами блокировок ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE и ACCESS EXCLUSIVE. Этот режим гарантирует, что кроме транзакции, получившей эту блокировку, никакая другая транзакция не может обращаться к таблице каким-либо способом.

Запрашивается командами DROP TABLE, TRUNCATE, REINDEX, CLUSTER, VACUUM FULL и REFRESH MATERIALIZED VIEW (без CONCURRENTLY). Блокировку на этом уровне запрашивают также многие виды ALTER TABLE. В этом режиме по умолчанию запрашивают блокировку и операторы LOCK TABLE, если явно не выбран другой режим.

> **Подсказка**
>
> Только блокировка ACCESS EXCLUSIVE блокирует оператор SELECT (без FOR UPDATE/SHARE).

Кроме того, для поддержки системы 1С:Предприятие реализованы ещё два режима блокировок. Эти режимы не конфликтуют ни с какими режимами, описанными выше. Их можно использовать, но лучше вместо них применять рекомендательные блокировки, которые предоставляют ту же функциональность.

APPLICATION SHARE

Конфликтует только с режимом блокировки APPLICATION EXCLUSIVE.

APPLICATION EXCLUSIVE

Конфликтует с режимами блокировок APPLICATION SHARE и APPLICATION EXCLUSIVE.

Полученная транзакцией блокировка обычно сохраняется до конца транзакции. Но если блокировка получена после установки точки сохранения, она освобождается немедленно в случае отката к этой точке. Это согласуется с принципом действия ROLLBACK — эта команда отменяет эффекты всех команд после точки сохранения. То же справедливо и для блокировок, полученных в блоке исключений PL/pgSQL: при выходе из блока с ошибкой такие блокировки освобождаются.

Таблица 13.2. Конфликтующие режимы блокировки

Запрашиваемый режим блокировки	Текущий режим блокировки									
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE	APPLICATION SHARE	APPLICATION EXCLUSIVE
ACCESS SHARE								X		
ROW SHARE							X	X		
ROW EXCLUSIVE					X	X	X	X		
SHARE UPDATE EXCLUSIVE				X	X	X	X	X		
SHARE			X	X		X	X	X		
SHARE ROW EXCLUSIVE			X	X	X	X	X	X		
EXCLUSIVE		X	X	X	X	X	X	X		
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X		
APPLICATION SHARE										X
APPLICATION EXCLUSIVE									X	X

13.3.2. Блокировки на уровне строк

Помимо блокировок на уровне таблицы, существуют блокировки на уровне строк, перечисленные ниже с контекстами, где Postgres Pro применяет их по умолчанию. Полный перечень конфликтов блокировок на уровне строк приведён в Таблице 13.3. Заметьте, что одна транзакция может владеть несколькими конфликтующими блокировками одной строки, даже в разных подтранзакциях; но две разных транзакции никогда не получат конфликтующие блокировки одной и той же строки. Блокировки на уровне строк блокируют только *запись в определённые строки*, но никак не влияют на выборку. Снимаются такие блокировки, как и блокировки на уровне таблицы, в конце транзакции или при откате к точке сохранения.

Режимы блокировки на уровне строк

`FOR UPDATE`

В режиме `FOR UPDATE` строки, выданные оператором `SELECT`, блокируются как для изменения. При этом они защищаются от блокировки, изменения и удаления другими транзакциями до завершения текущей. То есть другие транзакции, пытающиеся выполнить `UPDATE`, `DELETE`, `SELECT FOR UPDATE`, `SELECT FOR NO KEY UPDATE`, `SELECT FOR SHARE` или `SELECT FOR KEY SHARE` с этими строками, будут заблокированы до завершения текущей транзакции; и наоборот, команда `SELECT FOR UPDATE` будет ожидать окончания параллельной транзакции, в которой выполнилась одна из этих команд с той же строкой, а затем установит блокировку и вернёт изменённую строку (или не вернёт, если она была удалена). Однако в транзакции `REPEATABLE READ` или `SERIALIZABLE` возникнет ошибка, если блокируемая строка изменилась с момента начала транзакции. Подробнее это обсуждается в Разделе 13.4.

Режим блокировки `FOR UPDATE` также запрашивается на уровне строки любой командой `DELETE` и командой `UPDATE`, изменяющей значения определённых столбцов. В настоящее время

блокировка с UPDATE касается столбцов, по которым создан уникальный индекс, применимый в качестве внешнего ключа (так что на частичные индексы и индексы выражений это не распространяется), но в будущем это может поменяться.

FOR NO KEY UPDATE

Действует подобно FOR UPDATE, но запрашиваемая в этом режиме блокировка слабее: она не будет блокировать команды SELECT FOR KEY SHARE, пытающиеся получить блокировку тех же строк. Этот режим блокировки также запрашивается любой командой UPDATE, которая не требует блокировки FOR UPDATE.

FOR SHARE

Действует подобно FOR NO KEY UPDATE, за исключением того, что для каждой из полученных строк запрашивается разделяемая, а не исключительная блокировка. Разделяемая блокировка не позволяет другим транзакциям выполнять с этими строками UPDATE, DELETE, SELECT FOR UPDATE или SELECT FOR NO KEY UPDATE, но допускает SELECT FOR SHARE и SELECT FOR KEY SHARE.

FOR KEY SHARE

Действует подобно FOR SHARE, но устанавливает более слабую блокировку: блокируется SELECT FOR UPDATE, но не SELECT FOR NO KEY UPDATE. Блокировка разделяемого ключа не позволяет другим транзакциям выполнять команды DELETE и UPDATE, только если они меняют значение ключа (но не другие UPDATE), и при этом допускает выполнение команд SELECT FOR NO KEY UPDATE, SELECT FOR SHARE и SELECT FOR KEY SHARE.

Postgres Pro не держит информацию об изменённых строках в памяти, так что никаких ограничений на число блокируемых строк нет. Однако блокировка строки может повлечь запись на диск, например, если SELECT FOR UPDATE изменяет выбранные строки, чтобы заблокировать их, при этом происходит запись на диск.

Таблица 13.3. Конфликтующие блокировки на уровне строк

Запрашиваемый режим блокировки	**Текущий режим блокировки**			
	FOR KEY SHARE	**FOR SHARE**	**FOR NO KEY UPDATE**	**FOR UPDATE**
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

13.3.3. Блокировки на уровне страниц

В дополнение к блокировкам на уровне таблицы и строк, для управления доступом к страницам таблиц в общих буферах используются блокировки на уровне страниц, исключительные и разделяемые. Эти блокировки освобождаются немедленно после выборки или изменения строк. Разработчикам приложений обычно можно не задумываться о блокировках страниц, здесь они упоминаются только для полноты картины.

13.3.4. Взаимоблокировки

Частое применение явных блокировок может увеличить вероятность *взаимоблокировок*, то есть ситуаций, когда две (или более) транзакций держат блокировки так, что взаимно блокируют друг друга. Например, если транзакция 1 получает исключительную блокировку таблицы A, а затем пытается получить исключительную блокировку таблицы B, которую до этого получила транзакция 2, в данный момент требующая исключительную блокировку таблицы A, ни одна из транзакций не сможет продолжить работу. Postgres Pro автоматически выявляет такие ситуации и разрешает их, прерывая одну из сцепившихся транзакций и тем самым позволяя другой (другим)

продолжить работу. (Какая именно транзакция будет прервана, обычно сложно предсказать, так что рассчитывать на определённое поведение не следует.)

Заметьте, что взаимоблокировки могут вызываться и блокировками на уровне строк (таким образом, они возможны, даже если не применяются явные блокировки). Рассмотрим случай, когда две параллельных транзакции изменяют таблицу. Первая транзакция выполняет:

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;
```

При этом она получает блокировку строки с указанным номером счёта. Затем вторая транзакция выполняет:

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;
```

Первый оператор `UPDATE` успешно получает блокировку указанной строки и изменяет данные в ней. Однако второй оператор `UPDATE` обнаруживает, что строка, которую он пытается изменить, уже заблокирована, так что он ждёт завершения транзакции, получившей блокировку. Таким образом, вторая транзакция сможет продолжиться только после завершения первой. Теперь первая транзакция выполняет:

```
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;
```

Первая транзакция пытается получить блокировку заданной строки, но ей это не удаётся: эта блокировка уже принадлежит второй транзакции. Поэтому первой транзакции остаётся только ждать завершения второй. В результате первая транзакция блокируется второй, а вторая — первой: происходит взаимоблокировка. Postgres Pro выявляет эту ситуацию и прерывает одну из транзакций.

Обычно лучший способ предотвращения взаимоблокировок — добиться, чтобы все приложения, обращающиеся к базе данных, запрашивали блокировки нескольких объектов единообразно. В данном примере, если бы обе транзакции изменяли строки в одном порядке, взаимоблокировка бы не произошла. Блокировки в транзакции следует упорядочивать так, чтобы первой для какого-либо объекта запрашивалась наиболее ограничивающая из тех, которые для него потребуются. Если заранее обеспечить такой порядок нельзя, взаимоблокировки можно обработать по факту, повторяя прерванные транзакции.

Если ситуация взаимоблокировки не будет выявлена, транзакция, ожидающая блокировки на уровне таблицы или строки, будет ждать её освобождения неограниченное время. Это означает, что приложения не должны оставлять транзакции открытыми долгое время (например, ожидая ввода пользователя).

13.3.5. Рекомендательные блокировки

Postgres Pro также имеет средства создания блокировок, смысл которых определяют сами приложения. Такие блокировки называются *рекомендательными*, так как система не форсирует их использование — правильно их использовать должно само приложение. Рекомендательные блокировки бывают полезны для реализаций стратегий блокирования, плохо вписывающихся в модель MVCC. Например, рекомендательные блокировки часто применяются для исполнения стратегии пессимистичной блокировки, типичной для систем управления данными «плоский файл». Хотя для этого можно использовать и дополнительные флаги в таблицах, рекомендательные блокировки работают быстрее, не нагружают таблицы и автоматически ликвидируется сервером в конце сеанса.

В Postgres Pro есть два варианта получить рекомендательные блокировки: на уровне сеанса и на уровне транзакции. Рекомендательная блокировка, полученная на уровне сеанса, удерживается, пока она не будет явно освобождена, или до конца сеанса. В отличие от стандартных рекомендательные блокировки уровня сеанса нарушают логику транзакций — блокировка, полученная в транзакции, даже если произойдёт откат этой транзакции, будет сохраняться в сеансе; аналогично, освобождение блокировки остаётся в силе, даже если транзакция, в которой оно было выполнено, позже прерывается. Вызывающий процесс может запросить блокировку несколько раз; при этом каждому запросу блокировки должен соответствовать запрос освобождения, чтобы она была действительно освобождена. Рекомендательные блокировки на

уровне транзакций, напротив, во многом похожи на обычные блокировки: они автоматически освобождаются в конце транзакций и не требуют явного освобождения. Для кратковременного применения блокировок это поведение часто более уместно, чем поведение рекомендательных блокировок на уровне сеанса. Запросы рекомендательных блокировок одного идентификатора на уровне сеанса и на уровне транзакции будут блокировать друг друга вполне предсказуемым образом. Если сеанс уже владеет данной рекомендуемой блокировкой, дополнительные запросы её в том же сеансе будут всегда успешны, даже если её ожидают другие сеансы. Это утверждение справедливо вне зависимости от того, на каком уровне (сеанса или транзакции) установлены или запрашиваются новые блокировки.

Как и остальные блокировки в Postgres Pro, все рекомендательные блокировки, связанные с любыми сеансами, можно просмотреть в системном представлении `pg_locks`.

И рекомендательные, и обычные блокировки сохраняются в области общей памяти, размер которой определяется параметрами конфигурации max_locks_per_transaction и max_connections. Важно, чтобы этой памяти было достаточно, так как в противном случае сервер не сможет выдать никакую блокировку. Таким образом, число рекомендуемых блокировок, которые может выдать сервер, ограничивается обычно десятками или сотнями тысяч в зависимости от конфигурации сервера.

В определённых случаях при использовании рекомендательных блокировок, особенно в запросах с явными указаниями `ORDER BY` и `LIMIT`, важно учитывать, что получаемые блокировки могут зависеть от порядка вычисления SQL-выражений. Например:

```
SELECT pg_advisory_lock(id) FROM foo WHERE id = 12345; -- ok
SELECT pg_advisory_lock(id) FROM foo WHERE id > 12345 LIMIT 100; -- опасно!
SELECT pg_advisory_lock(q.id) FROM
(
  SELECT id FROM foo WHERE id > 12345 LIMIT 100
) q; -- ok
```

В этом примере второй вариант опасен, так как `LIMIT` не обязательно будет применяться перед вызовом функции блокировки. В результате приложение может получить блокировки, на которые оно не рассчитывает и которые оно не сможет освободить (до завершения сеанса). С точки зрения приложения такие блокировки окажутся в подвешенном состоянии, хотя они и будут отображаться в `pg_locks`.

Функции, предназначенные для работы с рекомендательными блокировками, описаны в Подразделе 9.26.10.

13.4. Проверки целостности данных на уровне приложения

Используя транзакции Read Committed, очень сложно обеспечить целостность данных с точки зрения бизнес-логики, так как представление данных смещается с каждым оператором и даже один оператор может не ограничиваться своим снимком состояния в случае конфликта записи.

Хотя транзакция Repeatable Read получает стабильное представление данных в процессе выполнения, с использованием снимков MVCC для проверки целостности данных всё же связаны тонкие моменты, включая так называемые *конфликты чтения/записи*. Если одна транзакция записывает данные, а другая в это же время пытается их прочитать (до или после записи), она не может увидеть результат работы первой. В таком случае создаётся впечатление, что читающая транзакция выполняется первой вне зависимости от того, какая из них была начата или зафиксирована раньше. Если этим всё и ограничивается, нет никаких проблем, но если читающая транзакция также пишет данные, которые читает параллельная транзакция, получается, что теперь эта транзакция будет исполняться, как будто она запущена перед другими вышеупомянутыми. Если же транзакция, которая должна исполняться как последняя, на самом деле зафиксирована первой, в графе упорядоченных транзакций легко может возникнуть цикл. И когда он возникает, проверки целостности не будут работать правильно без дополнительных мер.

Как было сказано в Подразделе 13.2.3, сериализуемые транзакции представляют собой те же транзакции Repeatable Read, но дополненные неблокирующим механизмом отслеживания опасных условий конфликтов чтения/записи. Когда выявляется условие, приводящее к циклу в порядке транзакций, одна из этих транзакций откатывается и этот цикл таким образом разрывается.

13.4.1. Обеспечение согласованности в сериализуемых транзакциях

Если для всех операций чтения и записи, нуждающихся в согласованном представлении данных, используются транзакции уровня изоляции Serializable, это обеспечивает необходимую согласованность без дополнительных усилий. Приложения из других окружений, применяющие сериализуемые транзакции для обеспечения целостности, в Postgres Pro в этом смысле будут «просто работать».

Применение этого подхода избавляет программистов приложений от лишних сложностей, если приложение использует инфраструктуру, которая автоматически повторяет транзакции в случае отката из-за сбоев сериализации. Возможно, `serializable` стоит даже установить в качестве уровня изоляции по умолчанию (`default_transaction_isolation`). Также имеет смысл принять меры для предотвращения использования других уровней изоляции, непреднамеренного или с целью обойти проверки целостности, например проверять уровень изоляции в триггерах.

Рекомендации по увеличению быстродействия приведены в Подразделе 13.2.3.

Предупреждение

Защита целостности с применением сериализуемых транзакций пока ещё не поддерживается в режиме горячего резерва (Раздел 25.5). Поэтому там, где применяется горячий резерв, следует использовать уровень Repeatable Read и явные блокировки на главном сервере.

13.4.2. Применение явных блокировок для обеспечения согласованности

Когда возможны несериализуемые операции записи, для обеспечения целостности строк и защиты от одновременных изменений, следует использовать `SELECT FOR UPDATE`, `SELECT FOR SHARE` или соответствующий оператор `LOCK TABLE`. (`SELECT FOR UPDATE` и `SELECT FOR SHARE` защищают от параллельных изменений только возвращаемые строки, тогда как `LOCK TABLE` блокирует всю таблицу.) Это следует учитывать, перенося в Postgres Pro приложения из других СУБД.

Мигрируя в Postgres Pro из других СУБД также следует учитывать, что команда `SELECT FOR UPDATE` сама по себе не гарантирует, что параллельная транзакция не изменит или не удалит выбранную строку. Для получения такой гарантии в Postgres Pro нужно именно изменить эту строку, даже если никакие значения в ней менять не требуется. `SELECT FOR UPDATE` *временно блокирует* другие транзакции, не давая им получить ту же блокировку или выполнить команды `UPDATE` или `DELETE`, которые бы повлияли на заблокированную строку, но как только транзакция, владеющая этой блокировкой, фиксируется или откатывается, заблокированная транзакция сможет выполнить конфликтующую операцию, если только для данной строки действительно не был выполнен `UPDATE`, пока транзакция владела блокировкой.

Реализация глобальной целостности с использованием несериализуемых транзакций MVCC требует более вдумчивого подхода. Например, банковскому приложению может потребоваться проверить, равняется ли сумма всех расходов в одной таблице сумме приходов в другой, при том, что обе таблицы активно изменяются. Просто сравнивать результаты двух успешных последовательных команд `SELECT sum(...)` в режиме Read Committed нельзя, так как вторая команда может захватить результаты транзакций, пропущенных первой. Подсчитывая суммы в одной транзакции Repeatable Read, можно получить точную картину только для транзакций, которые были зафиксированы до начала данной, но при этом может возникнуть законный

вопрос — будет ли этот результат актуален тогда, когда он будет выдан. Если транзакция Repeatable Read сама вносит какие-то изменения, прежде чем проверять равенство сумм, полезность этой проверки становится ещё более сомнительной, так как при проверке будут учитываться некоторые, но не все изменения, произошедшие после начала транзакции. В таких случаях предусмотрительный разработчик может заблокировать все таблицы, задействованные в проверке, чтобы получить картину действительности, не вызывающую сомнений. Для этого применяется блокировка `SHARE` (или более строгая), которая гарантирует, что в заблокированной таблице не будет незафиксированных изменений, за исключением тех, что внесла текущая транзакция.

Также заметьте, что, применяя явные блокировки для предотвращения параллельных операций записи, следует использовать либо режим Read Committed, либо в режиме Repeatable Read обязательно получать блокировки прежде, чем выполнять запросы. Блокировка, получаемая транзакцией Repeatable Read, гарантирует, что никакая другая транзакция, изменяющая таблицу, не выполняется, но если снимок состояния, полученный транзакцией, предшествует блокировке, он может не включать на данный момент уже зафиксированные изменения. Снимок состояния в транзакции Repeatable Read создаётся фактически на момент начала первой команды выборки или изменения данных (`SELECT`, `INSERT`, `UPDATE` или `DELETE`), так что получить явные блокировки можно до того, как он будет сформирован.

13.5. Ограничения

Некоторые команды DDL, в настоящее время это TRUNCATE и формы ALTER TABLE, перезаписывающие таблицу, не являются безопасными с точки зрения MVCC. Это значит, что после фиксации усечения или перезаписи таблица окажется пустой для всех параллельных транзакций, если они работают со снимком, полученным перед фиксацией такой команды DDL. Это может проявиться только в транзакции, которая не обращалась к таблице до момента начала команды DDL — любая транзакция, которая обращалась к ней раньше, получила бы как минимум блокировку `ACCESS SHARE`, которая заблокировала бы эту команду DDL до завершения транзакции. Поэтому такие команды не приводят ни к каким видимым несоответствиям с содержимым таблицы при последовательных запросах к целевой таблице, хотя возможно видимое несоответствие между содержимым целевой таблицы и другими таблицами в базе данных.

Поддержка уровня изоляции Serializable ещё не реализована для целевых серверов горячего резерва (они описываются в Разделе 25.5). На данный момент самый строгий уровень изоляции, поддерживаемый в режиме горячего резерва, это Repeatable Read. Хотя и тогда, когда главный сервер выполняет запись в транзакциях Serializable, все резервные серверы в итоге достигают согласованного состояния, но транзакция Repeatable Read на резервном сервере иногда может увидеть промежуточное состояние, не соответствующее результату последовательного выполнения транзакций на главном сервере.

Внутренние обращения к системным каталогам осуществляются за рамками уровня изоляции текущей транзакции. Это означает, что создаваемые объекты базы данных, например таблицы, оказываются видимыми для параллельных транзакций уровня Repeatable Read и Serializable, несмотря на то, что строки в них не видны. С другой стороны, запросы, обращающиеся к системным каталогам напрямую, не будут видеть строки, представляющие недавно созданные объекты базы данных, если эти запросы используют более высокие уровни изоляции.

13.6. Блокировки и индексы

Хотя Postgres Pro обеспечивает неблокирующий доступ на чтение/запись к данным таблиц, для индексов в настоящий момент это поддерживается не в полной мере. Postgres Pro управляет доступом к различным типам индексов следующим образом:

Индексы типа B-дерево, GiST и SP-GiST

> Для управления чтением/записью используются кратковременные блокировки на уровне страницы, исключительные и разделяемые. Блокировки освобождаются сразу после извлечения или добавления строки индекса. Эти типы индексов обеспечивают максимальное распараллеливание операций, не допуская взаимоблокировок.

Хеш-индексы

Для управления чтением/записью используются блокировки на уровне групп хеша. Блокировки освобождаются после обработки всей группы. Такие блокировки с точки зрения распараллеливания лучше, чем блокировки на уровне индекса, но не исключают взаимоблокировок, так как они сохраняются дольше, чем выполняется одна операция с индексом.

Индексы GIN

Для управления чтением/записью используются кратковременные блокировки на уровне страницы, исключительные и разделяемые. Блокировки освобождаются сразу после извлечения или добавления строки индекса. Но заметьте, что добавление значения в поле с GIN-индексом обычно влечёт добавление нескольких ключей индекса, так что GIN может проделывать целый ряд операций для одного значения.

В настоящее время в многопоточной среде наиболее производительны индексы-B-деревья; и так как они более функциональны, чем хеш-индексы, их рекомендуется использовать в такой среде для приложений, когда нужно индексировать скалярные данные. Если же нужно индексировать не скалярные данные, B-деревья не подходят, и вместо них следует использовать индексы GiST, SP-GiST или GIN.

Глава 14. Оптимизация производительности

Быстродействие запросов зависит от многих факторов. На некоторые из них могут воздействовать пользователи, а другие являются фундаментальными особенностями системы. В этой главе приводятся полезные советы, которые помогут понять их и оптимизировать производительность Postgres Pro.

14.1. Использование EXPLAIN

Выполняя любой полученный запрос, Postgres Pro разрабатывает для него *план запроса*. Выбор правильного плана, соответствующего структуре запроса и характеристикам данным, крайне важен для хорошей производительности, поэтому в системе работает сложный *планировщик*, задача которого — подобрать хороший план. Узнать, какой план был выбран для какого-либо запроса, можно с помощью команды EXPLAIN. Понимание плана — это искусство, и чтобы овладеть им, нужен определённый опыт, но этот раздел расскажет о самых простых вещах.

Приведённые ниже примеры показаны на тестовой базе данных, которая создаётся для выявления регрессий в исходных кодах PostgreSQL текущей версии. Для неё предварительно выполняется VACUUM ANALYZE. Вы должны получить похожие результаты, если возьмёте ту же базу данных и проделаете следующие действия, но примерная стоимость и ожидаемое число строк у вас может немного отличаться из-за того, что статистика команды ANALYZE рассчитывается по случайной выборке, а оценки стоимости зависят от конкретной платформы.

В этих примерах используется текстовый формат вывода EXPLAIN, принятый по умолчанию, как более компактный и удобный для восприятия человеком. Если вывод EXPLAIN нужно передать какой-либо программе для дальнейшего анализа, лучше использовать один из машинно-ориентированных форматов (XML, JSON или YAML).

14.1.1. Азы EXPLAIN

Структура плана запроса представляет собой дерево *узлов плана*. Узлы на нижнем уровне дерева — это узлы сканирования, которые возвращают необработанные данные таблицы. Разным типам доступа к таблице соответствуют разные узлы: последовательное сканирование, сканирование индекса и сканирование битовой карты. Источниками строк могут быть не только таблицы, но и например, предложения VALUES и функции, возвращающие множества во FROM, и они представляются отдельными типами узлов сканирования. Если запрос требует объединения, агрегатных вычислений, сортировки или других операций с исходными строками, над узлами сканирования появляются узлы, обозначающие эти операции. И так как обычно операции могут выполняться разными способами, на этом уровне тоже могут быть узлы разных типов. В выводе команды EXPLAIN для каждого узла в дереве плана отводится одна строка, где показывается базовый тип узла плюс оценка стоимости выполнения данного узла, которую сделал для него планировщик. Если для узла выводятся дополнительные свойства, в вывод могут добавляться дополнительные строки, с отступом от основной информации узла. В самой первой строке (основной строке самого верхнего узла) выводится общая стоимость выполнения для всего плана; именно это значение планировщик старается минимизировать.

Взгляните на следующий простейший пример, просто иллюстрирующий формат вывода:

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

Этот запрос не содержит предложения WHERE, поэтому он должен просканировать все строки таблицы, так что планировщик выбрал план простого последовательного сканирования. Числа, перечисленные в скобках (слева направо), имеют следующий смысл:

- Приблизительная стоимость запуска. Это время, которое проходит, прежде чем начнётся этап вывода данных, например для сортирующего узла это время сортировки.
- Приблизительная общая стоимость. Она вычисляется в предположении, что узел плана выполняется до конца, то есть возвращает все доступные строки. На практике родительский узел может досрочно прекратить чтение строк дочернего (см. приведённый ниже пример с `LIMIT`).
- Ожидаемое число строк, которое должен вывести этот узел плана. При этом так же предполагается, что узел выполняется до конца.
- Ожидаемый средний размер строк, выводимых этим узлом плана (в байтах).

Стоимость может измеряться в произвольных единицах, определяемых параметрами планировщика (см. Подраздел 18.7.2). Традиционно единицей стоимости считается операция чтения страницы с диска; то есть seq_page_cost обычно равен `1.0`, а другие параметры задаётся относительно него. Примеры в этом разделе выполняются со стандартными параметрами стоимости.

Важно понимать, что стоимость узла верхнего уровня включает стоимость всех его потомков. Также важно осознавать, что эта стоимость отражает только те факторы, которые учитывает планировщик. В частности, она не зависит от времени, необходимого для передачи результирующих строк клиенту, хотя оно может составлять значительную часть общего времени выполнения запроса. Тем не менее планировщик игнорирует эту величину, так как он всё равно не сможет изменить её, выбрав другой план. (Мы верим в то, что любой правильный план запроса выдаёт один и тот же набор строк.)

Значение `rows` здесь имеет особенность — оно выражает не число строк, обработанных или просканированных узлом плана, а число строк, выданных этим узлом. Часто оно окажется меньше числа просканированных строк в результате применённой к узлу фильтрации по условиям `WHERE`. В идеале, на верхнем уровне это значение будет приблизительно равно числу строк, которое фактически возвращает, изменяет или удаляет запрос.

Возвращаясь к нашему примеру:

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

Эти числа получаются очень просто. Выполните:

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

и вы увидите, что `tenk1` содержит 358 страниц диска и 10000 строк. Общая стоимость вычисляется как (число_чтений_диска * seq_page_cost) + (число_просканированных_строк * cpu_tuple_cost). По умолчанию, `seq_page_cost` равно 1.0, а `cpu_tuple_cost` — 0.01, так что приблизительная стоимость запроса равна (358 * 1.0) + (10000 * 0.01) = 458.

Теперь давайте изменим запрос, добавив в него предложение `WHERE`:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7001 width=244)
   Filter: (unique1 < 7000)
```

Заметьте, что в выводе `EXPLAIN` показано, что условие `WHERE` применено как «фильтр» к узлу плана Seq Scan (Последовательное сканирование). Это означает, что узел плана проверяет это условие для каждого просканированного им узла и выводит только те строки, которые удовлетворяют

ему. Предложение WHERE повлияло на оценку числа выходных строк. Однако при сканировании потребуется прочитать все 10000 строк, поэтому общая стоимость не уменьшилась. На деле она даже немного увеличилась (на 10000 * cpu_operator_cost, если быть точными), отражая дополнительное время, которое потребуется процессору на проверку условия WHERE.

Фактическое число строк результата этого запроса будет равно 7000, но значение rows даёт только приблизительное значение. Если вы попытаетесь повторить этот эксперимент, вы можете получить немного другую оценку; более того, она может меняться после каждой команды ANALYZE, так как ANALYZE получает статистику по случайной выборке таблицы.

Теперь давайте сделаем ограничение более избирательным:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

                            QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.07..229.20 rows=101 width=244)
   Recheck Cond: (unique1 < 100)
   ->  Bitmap Index Scan on tenk1_unique1
                                        (cost=0.00..5.01 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

В данном случае планировщик решил использовать план из двух этапов: сначала дочерний узел плана просматривает индекс и находит в нём адреса строк, соответствующих условию индекса, а затем верхний узел собственно выбирает эти строки из таблицы. Выбирать строки по отдельности гораздо дороже, чем просто читать их последовательно, но так как читать придётся не все страницы таблицы, это всё равно будет дешевле, чем сканировать всю таблицу. (Использование двух уровней плана объясняется тем, что верхний узел сортирует адреса строк, выбранных из индекса, в физическом порядке, прежде чем читать, чтобы снизить стоимость отдельных чтений. Слово «bitmap» (битовая карта) в имени узла обозначает механизм, выполняющий сортировку.)

Теперь давайте добавим ещё одно условие в предложение WHERE:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND stringu1 = 'xxx';

                            QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.01..229.40 rows=1 width=244)
   Recheck Cond: (unique1 < 100)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1
                                        (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

Добавленное условие stringu1 = 'xxx' уменьшает оценку числа результирующих строк, но не стоимость запроса, так как просматриваться будет тот же набор строк, что и раньше. Заметьте, что условие на stringu1 не добавляется в качестве условия индекса, так как индекс построен только по столбцу unique1. Вместо этого оно применяется как фильтр к строкам, полученным по индексу. В результате стоимость даже немного увеличилась, отражая добавление этой проверки.

В некоторых случаях планировщик предпочтёт «простой» план сканирования индекса:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 = 42;

                            QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1 (cost=0.29..8.30 rows=1 width=244)
   Index Cond: (unique1 = 42)
```

В плане такого типа строки таблицы выбираются в порядке индекса, в результате чего чтение их обходится дороже, но так как их немного, дополнительно сортировать положения строк не стоит.

Вы часто будете встречать этот тип плана в запросах, которые выбирают всего одну строку. Также он часто задействуется там, где условие `ORDER BY` соответствует порядку индекса, так как в этих случаях для выполнения `ORDER BY` не требуется дополнительный шаг сортировки.

Если в таблице есть отдельные индексы по разным столбцам, фигурирующим в `WHERE`, планировщик может выбрать сочетание этих индексов (с AND и OR):

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
               Index Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0)
               Index Cond: (unique2 > 9000)
```

Но для этого потребуется обойти оба индекса, так что это не обязательно будет выгоднее, чем просто просмотреть один индекс, а второе условие обработать как фильтр. Измените диапазон и вы увидите, как это повлияет на план.

Следующий пример иллюстрирует эффекты `LIMIT`:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Limit  (cost=0.29..14.48 rows=2 width=244)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..71.27 rows=10 width=244)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
```

Это тот же запрос, что и раньше, но добавили мы в него `LIMIT`, чтобы возвращались не все строки, и планировщик решает выполнять запрос по-другому. Заметьте, что общая стоимость и число строк для узла Index Scan рассчитываются в предположении, что он будет выполняться полностью. Однако узел Limit должен остановиться, получив только пятую часть всех строк, так что его стоимость будет составлять одну пятую от вычисленной ранее, и это и будет итоговой оценкой стоимости запроса. С другой стороны, планировщик мог бы просто добавить в предыдущий план узел Limit, но это не избавило бы от затрат на запуск сканирования битовой карты, а значит, общая стоимость была бы выше 25 единиц.

Давайте попробуем соединить две таблицы по столбцам, которые мы уже использовали:

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
--------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1 width=244)
         Index Cond: (unique2 = t1.unique2)
```

В этом плане появляется узел соединения с вложенным циклом, на вход которому поступают данные от двух его потомков, узлов сканирования. Эту структуру плана отражает отступ основных

строк его узлов. Первый, или «внешний», потомок соединения — узел сканирования битовой карты, похожий на те, что мы видели раньше. Его стоимость и число строк те же, что мы получили бы для запроса `SELECT ... WHERE unique1 < 10`, так как к этому узлу добавлено предложение `WHERE unique1 < 10`. Условие `t1.unique2 = t2.unique2` ещё не учитывается, поэтому оно не влияет на число строк узла внешнего сканирования. Узел соединения с вложенным циклом будет выполнять узел «внутреннего» потомка для каждой строки, полученной из внешнего потомка. Значения столбцов из текущей внешней строки могут использоваться во внутреннем сканировании (в данном случае это значение `t1.unique2`), поэтому мы получаем план и стоимость примерно такие, как и раньше для простого запроса `SELECT ... WHERE t2.unique2 = константа`. (На самом деле оценочная стоимость немного меньше, в предположении, что при неоднократном сканировании индекса по `t2` положительную роль сыграет кеширование.) В результате стоимость узла цикла складывается из стоимости внешнего сканирования, цены внутреннего сканирования, умноженной на число строк (здесь 10 * 7.91), и небольшой наценки за обработку соединения.

В этом примере число выходных строк соединения равно произведению чисел строк двух узлов сканирования, но это не всегда будет так, потому что в дополнительных условиях `WHERE` могут упоминаться обе таблицы, так что применить их можно будет только в точке соединения, а не в одном из узлов сканирования. Например:

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t2.unique2 < 10 AND t1.hundred < t2.hundred;

                                      QUERY PLAN
--------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..49.46 rows=33 width=488)
   Join Filter: (t1.hundred < t2.hundred)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Materialize  (cost=0.29..8.51 rows=10 width=244)
         ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..8.46 rows=10
 width=244)
               Index Cond: (unique2 < 10)
```

Условие `t1.hundred < t2.hundred` не может быть проверено в индексе `tenk2_unique2`, поэтому оно применяется в узле соединения. Это уменьшает оценку числа выходных строк, тогда как число строк в узлах сканирования не меняется.

Заметьте, что здесь планировщик решил «материализовать» внутреннее отношение соединения, поместив поверх него узел плана Materialize (Материализовать). Это значит, что сканирование индекса `t2` будет выполняться только единожды, при том, что узлу вложенного цикла соединения потребуется прочитать данные десять раз, по числу строк во внешнем соединении. Узел Materialize сохраняет считанные данные в памяти, чтобы затем выдать их из памяти на следующих проходах.

Выполняя внешние соединения, вы можете встретить узлы плана с присоединёнными условиями, как обычными «Filter», так и «Join Filter» (Фильтр соединения). Условия Join Filter формируются из предложения `ON` для внешнего соединения, так что если строка не удовлетворяет условию Join Filter, она всё же выдаётся как строка, дополненная значениями NULL. Обычное же условие Filter применяется после правил внешнего соединения и поэтому полностью исключает строки. Во внутреннем соединении оба этих фильтра работают одинаково.

Если немного изменить избирательность запроса, мы можем получить совсем другой план соединения:

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;
```

```
                                       QUERY PLAN
-----------------------------------------------------------------------------------------
 Hash Join  (cost=230.47..713.98 rows=101 width=488)
   Hash Cond: (t2.unique2 = t1.unique2)
   ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244)
   ->  Hash  (cost=229.20..229.20 rows=101 width=244)
         ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101 width=244)
               Recheck Cond: (unique1 < 100)
               ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101
 width=0)
                     Index Cond: (unique1 < 100)
```

Здесь планировщик выбирает соединение по хешу, при котором строки одной таблицы записываются в хеш-таблицу в памяти, после чего сканируется другая таблица и для каждой её строки проверяется соответствие по хеш-таблице. Обратите внимание, что и здесь отступы отражают структуру плана: результат сканирования битовой карты по `tenk1` подаётся на вход узлу Hash, который конструирует хеш-таблицу. Затем она передаётся узлу Hash Join, который читает строки из узла внешнего потомка и проверяет их по этой хеш-таблице.

Ещё один возможный тип соединения — соединение слиянием:

```
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;
```

```
                                       QUERY PLAN
-----------------------------------------------------------------------------------------
 Merge Join  (cost=198.11..268.19 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101
 width=244)
         Filter: (unique1 < 100)
   ->  Sort  (cost=197.83..200.33 rows=1000 width=244)
         Sort Key: t2.unique2
         ->  Seq Scan on onek t2  (cost=0.00..148.00 rows=1000 width=244)
```

Соединение слиянием требует, чтобы входные данные для него были отсортированы по ключам соединения. В этом плане данные `tenk1` сортируются после сканирования индекса, при котором все строки просматриваются в правильном порядке, но таблицу `onek` выгоднее оказывается последовательно просканировать и отсортировать, так как в этой таблице нужно обработать гораздо больше строк. (Последовательное сканирование и сортировка часто бывает быстрее сканирования индекса, когда нужно отсортировать много строк, так как при сканировании по индексу обращения к диску не упорядочены.)

Один из способов посмотреть различные планы — принудить планировщик не считать выбранную им стратегию самой выгодной, используя флаги, описанные в Подразделе 18.7.1. (Это полезный, хотя и грубый инструмент. См. также Раздел 14.3.) Например, если мы убеждены, что последовательное сканирование и сортировка — не лучший способ обработать таблицу `onek` в предыдущем примере, мы можем попробовать

```
SET enable_sort = off;

EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;
```

```
                                       QUERY PLAN
-----------------------------------------------------------------------------------------
 Merge Join  (cost=0.56..292.65 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
```

```
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101
 width=244)
         Filter: (unique1 < 100)
   ->  Index Scan using onek_unique2 on onek t2  (cost=0.28..224.79 rows=1000
 width=244)
```

Видно, что планировщик считает сортировку `onek` со сканированием индекса примерно на 12% дороже, чем последовательное сканирование и сортировку. Конечно, может возникнуть вопрос — а правильно ли это? Мы можем ответить на него, используя описанную ниже команду `EXPLAIN ANALYZE`.

14.1.2. EXPLAIN ANALYZE

Точность оценок планировщика можно проверить, используя команду `EXPLAIN` с параметром `ANALYZE`. С этим параметром `EXPLAIN` на самом деле выполняет запрос, а затем выводит фактическое число строк и время выполнения, накопленное в каждом узле плана, вместе с теми же оценками, что выдаёт обычная команда `EXPLAIN`. Например, мы можем получить примерно такой результат:

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488) (actual time=0.128..0.377 rows=10
 loops=1)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244) (actual
 time=0.057..0.121 rows=10 loops=1)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
 (actual time=0.024..0.024 rows=10 loops=1)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1 width=244)
 (actual time=0.021..0.022 rows=1 loops=10)
         Index Cond: (unique2 = t1.unique2)
 Planning time: 0.181 ms
 Execution time: 0.501 ms
```

Заметьте, что значения «actual time» (фактическое время) приводятся в миллисекундах, тогда как оценки `cost` (стоимость) выражаются в произвольных единицах, так что они вряд ли совпадут. Обычно важнее определить, насколько приблизительная оценка числа строк близка к действительности. В этом примере они в точности совпали, но на практике так бывает редко.

В некоторых планах запросов некоторый внутренний узел может выполняться неоднократно. Например, внутреннее сканирование индекса будет выполняться для каждой внешней строки во вложенном цикле верхнего уровня. В таких случаях значение `loops` (циклы) показывает, сколько всего раз выполнялся этот узел, а фактическое время и число строк вычисляется как среднее по всем итерациям. Это делается для того, чтобы полученные значения можно было сравнить с выводимыми приблизительными оценками. Чтобы получить общее время, затраченное на выполнение узла, время одной итерации нужно умножить на значение `loops`. В показанном выше примере мы потратили в общей сложности 0.220 мс на сканирование индекса в `tenk2`.

В ряде случаев `EXPLAIN ANALYZE` выводит дополнительную статистику по выполнению, включающую не только время выполнения узлов и число строк. Для узлов Sort и Hash, например выводится следующая информация:

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY t1.fivethous;
```

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Sort  (cost=717.34..717.59 rows=101 width=488) (actual time=7.761..7.774 rows=100
 loops=1)
   Sort Key: t1.fivethous
   Sort Method: quicksort  Memory: 77kB
   ->  Hash Join  (cost=230.47..713.98 rows=101 width=488) (actual time=0.711..7.427
 rows=100 loops=1)
         Hash Cond: (t2.unique2 = t1.unique2)
         ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244) (actual
 time=0.007..2.583 rows=10000 loops=1)
         ->  Hash  (cost=229.20..229.20 rows=101 width=244) (actual time=0.659..0.659
 rows=100 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 28kB
               ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101 width=244)
 (actual time=0.080..0.526 rows=100 loops=1)
                     Recheck Cond: (unique1 < 100)
                     ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101
 width=0) (actual time=0.049..0.049 rows=100 loops=1)
                           Index Cond: (unique1 < 100)
 Planning time: 0.194 ms
 Execution time: 8.008 ms
```

Для узла Sort показывается использованный метод и место сортировки (в памяти или на диске), а также задействованный объём памяти. Для узла Hash выводится число групп и пакетов хеша, а также максимальный объём, который заняла в памяти хеш-таблица. (Если число пакетов больше одного, часть хеш-таблицы будет выгружаться на диск и занимать какое-то пространство, но его объём здесь не показывается.)

Другая полезная дополнительная информация — число строк, удалённых условием фильтра:

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE ten < 7;

                                                  QUERY PLAN
-------------------------------------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7000 width=244) (actual time=0.016..5.107
 rows=7000 loops=1)
   Filter: (ten < 7)
   Rows Removed by Filter: 3000
 Planning time: 0.083 ms
 Execution time: 5.905 ms
```

Эти значения могут быть особенно ценны для условий фильтра, применённых к узлам соединения. Строка «Rows Removed» выводится, только когда условие фильтра отбрасывает минимум одну просканированную строку или потенциальную пару соединения, если это узел соединения.

Похожую ситуацию можно наблюдать при сканировании «неточного» индекса. Например, рассмотрим этот план поиска многоугольников, содержащих указанную точку:

```
EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                                  QUERY PLAN
-------------------------------------------------------------------------------------------
 Seq Scan on polygon_tbl  (cost=0.00..1.05 rows=1 width=32) (actual time=0.044..0.044
 rows=0 loops=1)
   Filter: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Filter: 4
 Planning time: 0.040 ms
 Execution time: 0.083 ms
```

Планировщик полагает (и вполне справедливо), что таблица слишком мала для сканирования по индексу, поэтому он выбирает последовательное сканирование, при котором все строки отбрасываются условием фильтра. Но если мы принудим его выбрать сканирование по индексу, мы получим:

```
SET enable_seqscan TO off;

EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Index Scan using gpolygonind on polygon_tbl  (cost=0.13..8.15 rows=1 width=32) (actual
 time=0.062..0.062 rows=0 loops=1)
   Index Cond: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Index Recheck: 1
 Planning time: 0.034 ms
 Execution time: 0.144 ms
```

Здесь мы видим, что индекс вернул одну потенциально подходящую строку, но затем она была отброшена при перепроверке условия индекса. Это объясняется тем, что индекс GiST является «неточным» для проверок включений многоугольников: фактически он возвращает строки с многоугольниками, перекрывающими точку по координатам, а затем для этих строк нужно выполнять точную проверку.

`EXPLAIN` принимает параметр `BUFFERS` (который также можно применять с `ANALYZE`), включающий ещё более подробную статистику выполнения запроса:

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                                           QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244) (actual
 time=0.323..0.342 rows=10 loops=1)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   Buffers: shared hit=15
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0) (actual time=0.309..0.309 rows=0
 loops=1)
         Buffers: shared hit=7
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
 (actual time=0.043..0.043 rows=100 loops=1)
               Index Cond: (unique1 < 100)
               Buffers: shared hit=2
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0)
 (actual time=0.227..0.227 rows=999 loops=1)
               Index Cond: (unique2 > 9000)
               Buffers: shared hit=5
 Planning time: 0.088 ms
 Execution time: 0.423 ms
```

Значения, которые выводятся с параметром `BUFFERS`, помогают понять, на какие части запроса приходится большинство операций ввода-вывода.

Не забывайте, что `EXPLAIN ANALYZE` действительно выполняет запрос, хотя его результаты могут не показываться, а заменяться выводом команды `EXPLAIN`. Поэтому при таком анализе возможны побочные эффекты. Если вы хотите проанализировать запрос, изменяющий данные, но при этом сохранить прежние данные таблицы, вы можете откатить транзакцию после запроса:

```
BEGIN;

EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1 WHERE unique1 < 100;
```

```
                                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Update on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual time=14.628..14.628
 rows=0 loops=1)
   ->  Bitmap Heap Scan on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual
 time=0.101..0.439 rows=100 loops=1)
         Recheck Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
 (actual time=0.043..0.043 rows=100 loops=1)
               Index Cond: (unique1 < 100)
 Planning time: 0.079 ms
 Execution time: 14.727 ms

ROLLBACK;
```

Как показано в этом примере, когда выполняется команда `INSERT`, `UPDATE` или `DELETE`, собственно изменение данных в таблице происходит в узле верхнего уровня Insert, Update или Delete. Узлы плана более низких уровней выполняют работу по нахождению старых строк и/или вычислению новых данных. Поэтому вверху мы видим тот же тип сканирования битовой карты, что и раньше, только теперь его вывод подаётся узлу Update, который сохраняет изменённые строки. Стоит отметить, что узел, изменяющий данные, может выполняться значительное время (в данном случае это составляет львиную часть всего времени), но планировщик не учитывает эту работу в оценке общей стоимости. Это связано с тем, что эта работа будет одинаковой при любом правильном плане запроса, и поэтому на выбор плана она не влияет.

Когда команда `UPDATE` или `DELETE` имеет дело с иерархией наследования, вывод может быть таким:

```
EXPLAIN UPDATE parent SET f2 = f2 + 1 WHERE f1 = 101;
                                      QUERY PLAN
--------------------------------------------------------------------------------
 Update on parent  (cost=0.00..24.53 rows=4 width=14)
   Update on parent
   Update on child1
   Update on child2
   Update on child3
   ->  Seq Scan on parent  (cost=0.00..0.00 rows=1 width=14)
         Filter: (f1 = 101)
   ->  Index Scan using child1_f1_key on child1  (cost=0.15..8.17 rows=1 width=14)
         Index Cond: (f1 = 101)
   ->  Index Scan using child2_f1_key on child2  (cost=0.15..8.17 rows=1 width=14)
         Index Cond: (f1 = 101)
   ->  Index Scan using child3_f1_key on child3  (cost=0.15..8.17 rows=1 width=14)
         Index Cond: (f1 = 101)
```

В этом примере узлу Update помимо изначально упомянутой в запросе родительской таблицы нужно обработать ещё три дочерние таблицы. Поэтому формируются четыре плана сканирования, по одному для каждой таблицы. Ясности ради для узла Update добавляется примечание, показывающее, какие именно таблицы будут изменяться, в том же порядке, в каком они идут в соответствующих внутренних планах. (Эти примечания появились в PostgreSQL 9.5; до этого о целевых таблицах приходилось догадываться, изучая внутренние планы узла.)

Под заголовком `Planning time` (Время планирования) команда `EXPLAIN ANALYZE` выводит время, затраченное на построение плана запроса из разобранного запроса и его оптимизацию. Время собственно разбора или перезаписи запроса в него не включается.

Значение `Execution time` (Время выполнения), выводимое командой `EXPLAIN ANALYZE`, включает продолжительность запуска и остановки исполнителя запроса, а также время выполнения всех сработавших триггеров, но не включает время разбора, перезаписи и планирования запроса. Время, потраченное на выполнение триггеров `BEFORE` (если такие имеются) включается во время соответствующих узлов Insert, Update или Delete node; но время выполнения триггеров `AFTER` не

учитывается, так как триггеры AFTER срабатывают после выполнения всего плана. Общее время, проведённое в каждом триггере (BEFORE или AFTER), также выводится отдельно. Заметьте, что триггеры отложенных ограничений выполняются только в конце транзакции, так что время их выполнения EXPLAIN ANALYZE не учитывает.

14.1.3. Ограничения

Время выполнения, измеренное командой EXPLAIN ANALYZE, может значительно отличаться от времени выполнения того же запроса в обычном режиме. Тому есть две основных причины. Во-первых, так как при анализе никакие строки результата не передаются клиенту, время ввода/вывода и передачи по сети не учитывается. Во-вторых, может быть существенной дополнительная нагрузка, связанная с функциями измерений EXPLAIN ANALYZE, особенно в системах, где вызов gettimeofday() выполняется медленно. Для измерения этой нагрузки вы можете воспользоваться утилитой pg_test_timing.

Результаты EXPLAIN не следует распространять на ситуации, значительно отличающиеся от тех, в которых вы проводите тестирование. В частности, не следует полагать, что выводы, полученные для игрушечной таблицы, будут применимы и для настоящих больших таблиц. Оценки стоимости нелинейны и планировщик может выбирать разные планы в зависимости от размера таблицы. Например, в крайнем случае вся таблица может уместиться в одну страницу диска, и тогда вы почти наверняка получите план последовательного сканирования, независимо от того, есть у неё и индексы или нет. Планировщик понимает, что для обработки таблицы ему в любом случае потребуется прочитать одну страницу, так что нет никакого смысла обращаться к ещё одной странице за индексом. (Мы наблюдали это в показанном выше примере с polygon_tbl.)

Бывает, что фактическое и приближённо оценённое значения не совпадают, но в этом нет ничего плохого. Например, это возможно, когда выполнение плана узла прекращается преждевременно из-за указания LIMIT или подобного эффекта. Например, для запроса с LIMIT, который мы пробовали раньше:

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------
 Limit  (cost=0.29..14.71 rows=2 width=244) (actual time=0.177..0.249 rows=2 loops=1)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..72.42 rows=10 width=244)
 (actual time=0.174..0.244 rows=2 loops=1)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
         Rows Removed by Filter: 287
 Planning time: 0.096 ms
 Execution time: 0.336 ms
```

Оценки стоимости и числа строк для узла Index Scan показываются в предположении, что этот узел будет выполняться до конца. Но в действительности узел Limit прекратил запрашивать строки, как только получил первые две, так что фактическое число строк равно 2 и время выполнения запроса будет меньше, чем рассчитал планировщик. Но это не ошибка, а просто следствие того, что оценённые и фактические значения выводятся по-разному.

Соединения слиянием также имеют свои особенности, которые могут ввести в заблуждение. Соединение слиянием прекратит читать один источник данных, если второй будет прочитан до конца, а следующее значение ключа в первом больше последнего значения во втором. В этом случае пар строк больше не будет, так что сканировать первый источник дальше нет смысла. В результате будут прочитаны не все строки одного потомка и вы получите тот же эффект, что и с LIMIT. Кроме того, если внешний (первый) потомок содержит строки с повторяющимися значениями ключа, внутренний (второй) потомок сдвинется назад и повторно выдаст строки для этого значения ключа. EXPLAIN ANALYZE считает эти повторяющиеся строки, как если бы это действительно были дополнительные строки внутреннего источника. Когда во внешнем узле много таких повторений ключей, фактическое число строк, подсчитанное для внутреннего узла, может значительно превышать число строк в соответствующей таблице.

Для узлов BitmapAnd (Логическое произведение битовых карт) и BitmapOr (Логическое сложение битовых карт) фактическое число строк всегда равно 0 из-за ограничений реализации.

14.2. Статистика, используемая планировщиком

14.2.1. Статистика по одному столбцу

Как было показано в предыдущем разделе, планировщик запросов должен оценить число строк, возвращаемых запросов, чтобы сделать правильный выбор в отношении плана запроса. В этом разделе кратко описывается статистика, которую использует система для этих оценок.

В частности, статистика включает общее число записей в каждой таблице и индексе, а также число дисковых блоков, которые они занимают. Эта информация содержится в таблице pg_class, в столбцах reltuples и relpages. Получить её можно, например так:

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'tenk1%';

       relname        | relkind | reltuples | relpages
----------------------+---------+-----------+----------
 tenk1                | r       |     10000 |      358
 tenk1_hundred        | i       |     10000 |       30
 tenk1_thous_tenthous | i       |     10000 |       30
 tenk1_unique1        | i       |     10000 |       30
 tenk1_unique2        | i       |     10000 |       30
(5 rows)
```

Здесь мы видим, что tenk1 содержит 10000 строк данных и столько же строк в индексах (что неудивительно), но объём индексов гораздо меньше таблицы.

Для большей эффективности reltuples и relpages не пересчитываются «на лету», так что они обычно содержат несколько устаревшие значения. Их обновляют команды VACUUM, ANALYZE и несколько команд DDL, такие как CREATE INDEX. VACUUM и ANALYZE могут не сканировать всю таблицу (и обычно так и делают), а только вычислить приращение reltuples по части таблицы, так что результат остаётся приблизительным. В любом случае планировщик пересчитывает значения, полученные из pg_class, в пропорции к текущему физическому размеру таблицы и таким образом уточняет приближение.

Большинство запросов возвращают не все строки таблицы, а только немногие из них, ограниченные условиями WHERE. Поэтому планировщику нужно оценить *избирательность* условий WHERE, то есть определить, какой процент строк будет соответствовать каждому условию в предложении WHERE. Нужная для этого информация хранится в системном каталоге pg_statistic. Значения в pg_statistic обновляются командами ANALYZE и VACUUM ANALYZE и никогда не бывают точными, даже сразу после обновления.

Для исследования статистики лучше обращаться не непосредственно к таблице pg_statistic, а к представлению pg_stats, предназначенному для облегчения восприятия этой информации. Кроме того, представление pg_stats доступно для чтения всем, тогда как pg_statistic — только суперпользователям. (Это сделано для того, чтобы непривилегированные пользователи не могли ничего узнать о содержимом таблиц других людей из статистики. Представление pg_stats устроено так, что оно показывает строки только для тех таблиц, которые может читать данный пользователь.) Например, мы можем выполнить:

```
SELECT attname, inherited, n_distinct,
       array_to_string(most_common_vals, E'\n') as most_common_vals
FROM pg_stats
WHERE tablename = 'road';
```

```
 attname | inherited | n_distinct |          most_common_vals
---------+-----------+------------+------------------------------------
 name    | f         |  -0.363388 | I- 580                          Ramp+
         |           |            | I- 880                          Ramp+
         |           |            | Sp Railroad                         +
         |           |            | I- 580                              +
         |           |            | I- 680                          Ramp
 name    | t         |  -0.284859 | I- 880                          Ramp+
         |           |            | I- 580                          Ramp+
         |           |            | I- 680                          Ramp+
         |           |            | I- 580                              +
         |           |            | State Hwy 13                    Ramp
(2 rows)
```

Заметьте, что для одного столбца показываются две строки: одна соответствует полной иерархии наследования, построенной для таблицы `road` (`inherited=t`), и другая относится непосредственно к таблице `road` (`inherited=f`).

Объём информации, сохраняемой в `pg_statistic` командой `ANALYZE`, в частности максимальное число записей в массивах `most_common_vals` (самые популярные значения) и `histogram_bounds` (границы гистограмм) для каждого столбца, можно ограничить на уровне столбцов с помощью команды `ALTER TABLE SET STATISTICS` или глобально, установив параметр конфигурации default_statistics_target. В настоящее время ограничение по умолчанию равно 100 записям. Увеличивая этот предел, можно увеличить точность оценок планировщика, особенно для столбцов с нерегулярным распределением данных, ценой большего объёма `pg_statistic` и, возможно, увеличения времени расчёта этой статистики. И напротив, для столбцов с простым распределением данных может быть достаточно меньшего предела.

Подробнее использование статистики планировщиком описывается в Главе 65.

14.2.2. Расширенная статистика

Часто наблюдается картина, когда медленное выполнение запросов объясняется плохим выбором плана из-за того, что несколько столбцов, фигурирующих в условиях запроса, оказываются связанными. Обычно планировщик полагает, что несколько условий не зависят друг от друга, а это предположение оказывается неверным, когда значения этих столбцов коррелируют. Обычная статистика, которая по природе своей строится по отдельным столбцам, не может выявить корреляции между столбцами. Однако Postgres Pro имеет возможность вычислять *многовариантную статистику*, которая может собирать необходимую для этого информацию.

Так как число возможных комбинаций столбцов очень велико, автоматически вычислять многовариантную статистику непрактично. Вместо этого можно создать *объекты расширенной статистики*, чаще называемые просто *объектами статистики*, чтобы сервер собирал статистику по некоторым наборам столбцов, представляющим интерес.

Объекты статистики создаются командой CREATE STATISTICS. При создании такого объекта просто добавляется запись в каталоге, выражающая востребованность этой статистики. Собственно сбор данных выполняется процедурой `ANALYZE` (запускаемой вручную или автоматически в фоновом процессе). Изучить собранные значения можно в каталоге `pg_statistic_ext`.

Команда `ANALYZE` вычисляет расширенную статистику по той же выборке строк таблицы, которая используется и для вычисления обычной статистики по отдельным столбцам. Так как размер выборки увеличивается с увеличением целевого ограничения статистики для таблицы или любых её столбцов (как описано в предыдущем разделе), при большем целевом ограничении обычно получается более точная расширенная статистика, но и времени на её вычисление требуется больше.

В следующих подразделах описываются виды расширенной статистики, поддерживаемые в настоящее время.

14.2.2.1. Функциональные зависимости

Простейший вид расширенной статистики отслеживает *функциональные зависимости* (это понятие используется в определении нормальных форм баз данных). Мы называем столбец `b` функционально зависимым от столбца `a`, если знания значения `a` достаточно для определения значения `b`, то есть не существует двух строк с одинаковыми значениями `a`, но разными значениями `b`. В полностью нормализованной базе данных функциональные зависимости должны существовать только в первичных ключах и суперключах. Однако на практике многие наборы данных не нормализуются полностью по разным причинам; например, денормализация часто производится намеренно по соображениям производительности.

Существование функциональных зависимостей напрямую влияет на точность оценок в определённых запросах. Если запрос содержит условия как по независимым, так и по зависимым столбцам, условия по зависимым столбцам дополнительно не сокращают размер результата. Однако без знания о функциональной зависимости планировщик запросов будет полагать, что все условия независимы, и недооценит размер результата.

Для информирования планировщика о функциональных зависимостях команда `ANALYZE` может собирать показатели зависимостей между столбцами. Оценить степень зависимости между всеми наборами столбцов обошлось бы непозволительно дорого, поэтому сбор данных ограничивается только теми группами столбцов, которые фигурируют вместе в объекте статистики, определённом со свойством `dependencies`. Во избежание ненужных издержек при выполнении `ANALYZE` и последующем планировании запросов статистику с `dependencies` рекомендуется создавать только для групп сильно коррелирующих столбцов.

Взгляните на пример сбора статистики функциональной зависимости:

```
CREATE STATISTICS stts (dependencies) ON zip, city FROM zipcodes;

ANALYZE zipcodes;

SELECT stxname, stxkeys, stxdependencies
  FROM pg_statistic_ext
  WHERE stxname = 'stts';
 stxname | stxkeys |             stxdependencies
---------+---------+-----------------------------------------
 stts    | 1 5     | {"1 => 5": 1.000000, "5 => 1": 0.423130}
(1 row)
```

В показанном случае столбец 1 (код zip) полностью определяет столбец 5 (city), так что коэффициент равен 1.0, тогда как город (столбец city) определяет код ZIP только в 42% всех случаев, что означает, что многие города (58%) представлены несколькими кодами ZIP.

При вычислении избирательности запроса, в котором задействованы функционально зависимые столбцы, планировщик корректирует оценки избирательности по условиям, используя коэффициенты зависимостей, чтобы не допустить недооценки размера результата.

14.2.2.1.1. Ограничения функциональных зависимостей

Функциональные зависимости в настоящее время применяются только при рассмотрении простых условий с равенствами, сравнивающих значения столбцов с константами. Они не используются для улучшения оценок при проверке равенства двух столбцов или сравнении столбца с выражением, а также в условиях с диапазоном, условиях `LIKE` или любых других видах условий.

Рассматривая функциональные зависимости, планировщик предполагает, что условия по задействованным столбцам совместимы и таким образом избыточны. Если условия несовместимы, правильной оценкой должен быть ноль строк, но эта возможность не рассматривается. Например, с таким запросом

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip = '94105';
```

планировщик отбросит условие с `city`, так как оно не влияет на избирательность, что верно. Однако он сделает то же предположение и в таком случае:

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip = '90210';
```

хотя на самом деле этому запросу будет удовлетворять ноль строк. Но статистика функциональной зависимости не даёт достаточно информации, чтобы прийти к такому заключению.

Во многих практических ситуациях это предположение обычно удовлетворяется; например, графический интерфейс приложения для последующего формирования запроса может не допускать выбор несовместимого сочетания города и кода ZIP. Но когда это не так, статистика функциональной зависимости может не подойти.

14.2.2.2. Многовариантное число различных значений

Статистика по одному столбцу содержит число различных значений в каждом отдельном столбце. Оценки числа различных значений в сочетании нескольких столбцов (например, в `GROUP BY a, b`) часто оказываются ошибочными, когда планировщик имеет статистические данные только по отдельным столбцам, что приводит к выбору плохих планов.

Для улучшения таких оценок операция `ANALYZE` может собирать статистику по различным значениям для группы столбцов. Как и ранее, это непрактично делать для каждой возможной группы столбцов, так что данные собираются только по тем группам столбцов, которые указаны в определении объекта статистики, создаваемого со свойством `ndistinct`. Данные будут собираться по всем возможным сочетаниям из двух или нескольких столбцов из перечисленных в определении.

В продолжение предыдущего примера, количества различных значений в таблице ZIP-кодов могут выглядеть так:

```
CREATE STATISTICS stts2 (ndistinct) ON zip, state, city FROM zipcodes;

ANALYZE zipcodes;

SELECT stxkeys AS k, stxndistinct AS nd
  FROM pg_statistic_ext
  WHERE stxname = 'stts2';
-[ RECORD 1 ]-----------------------------------------------------------
k  | 1 2 5
nd | {"1, 2": 33178, "1, 5": 33178, "2, 5": 27435, "1, 2, 5": 33178}
(1 row)
```

Как видно, есть три комбинации столбцов, имеющих 33178 различных значений: код ZIP и штат; код ZIP и город; код ZIP, город и штат (то, что все эти числа равны, ожидаемый факт, так как сам по себе код ZIP в этой таблице уникален). С другой стороны, сочетание города и штата даёт только 27435 различных значений.

Объект статистики `ndistinct` рекомендуется создавать только для тех сочетаний столбцов, которые действительно используются при группировке, и только когда неправильная оценка числа групп может привести к выбору плохих планов. В противном случае усилия, потраченные на выполнение `ANALYZE`, будут напрасными.

14.3. Управление планировщиком с помощью явных предложений `JOIN`

Поведением планировщика в некоторой степени можно управлять, используя явный синтаксис `JOIN`. Понять, когда и почему это бывает нужно, поможет небольшое введение.

В простом запросе с соединением, например таком:

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

планировщик может соединять данные таблицы в любом порядке. Например, он может разработать план, в котором сначала A соединяется с B по условию `WHERE a.id = b.id`, а затем C

соединяется с получившейся таблицей по другому условию WHERE. Либо он может соединить B с C, а затем с A результатом соединения. Он также может соединить сначала A с C, а затем результат с B — но это будет не эффективно, так как ему придётся сформировать полное декартово произведение A и C из-за отсутствия в предложении WHERE условия, подходящего для оптимизации соединения. (В Postgres Pro исполнитель запросов может соединять только по две таблицы, поэтому для получения результата нужно выбрать один из этих способов.) При этом важно понимать, что все эти разные способы соединения дают одинаковые по смыслу результаты, но стоимость их может различаться многократно. Поэтому планировщик должен изучить их все и найти самый эффективный способ выполнения запроса.

Когда запрос включает только две или три таблицы, возможны всего несколько вариантов их соединения. Но их число растёт экспоненциально с увеличением числа задействованных таблиц. Если число таблиц больше десяти, уже практически невозможно выполнить полный перебор всех вариантов, и даже для шести или семи таблиц планирование может занять недопустимо много времени. Когда таблиц слишком много, планировщик Postgres Pro переключается с полного поиска на алгоритм *генетического* вероятностного поиска в ограниченном числе вариантов. (Порог для этого переключения задаётся параметром выполнения geqo_threshold.) Генетический поиск выполняется быстрее, но не гарантирует, что найденный план будет наилучшим.

Когда запрос включает внешние соединения, планировщик имеет меньше степеней свободы, чем с обычными (внутренними) соединениями. Например, рассмотрим запрос:

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Хотя ограничения в этом запросе очень похожи на показанные в предыдущем примере, смысл его отличается, так как результирующая строка должна выдаваться для каждой строки A, даже если для неё не находится соответствия в соединении B и C. Таким образом, здесь планировщик не может выбирать порядок соединения: он должен соединить B с C, а затем соединить A с результатом. Соответственно, и план этого запроса построится быстрее, чем предыдущего. В других случаях планировщик сможет определить, что можно безопасно выбрать один из нескольких способов соединения. Например, для запроса:

```
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);
```

можно соединить A либо с B, либо с C. В настоящее время только FULL JOIN полностью ограничивает порядок соединения. На практике в большинстве запросов с LEFT JOIN и RIGHT JOIN порядком можно управлять в некоторой степени.

Синтаксис явного внутреннего соединения (INNER JOIN, CROSS JOIN или лаконичный JOIN) по смыслу равнозначен перечислению отношений в предложении FROM, так что он никак не ограничивает порядок соединений.

Хотя большинство видов JOIN не полностью ограничивают порядок соединения, в Postgres Pro можно принудить планировщик обрабатывать все предложения JOIN как ограничивающие этот порядок. Например, следующие три запроса логически равнозначны:

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Но если мы укажем планировщику соблюдать порядок JOIN, на планирование второго и третьего уйдёт меньше времени. Когда речь идёт только о трёх таблицах, выигрыш будет незначительным, но для множества таблиц это может быть очень эффективно.

Чтобы планировщик соблюдал порядок внутреннего соединения, выраженный явно предложениями JOIN, нужно присвоить параметру выполнения join_collapse_limit значение 1. (Другие допустимые значения обсуждаются ниже.)

Чтобы сократить время поиска, необязательно полностью ограничивать порядок соединений, в JOIN можно соединять элементы как в обычном списке FROM. Например, рассмотрите следующий запрос:

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

Если join_collapse_limit = 1, планировщик будет вынужден соединить A с B раньше, чем результат с другими таблицами, но в дальнейшем выборе вариантов он не ограничен. В данном примере число возможных вариантов соединения уменьшается в 5 раз.

Упрощать для планировщика задачу перебора вариантов таким способом — это полезный приём, помогающий не только выбрать сократить время планирования, но и подтолкнуть планировщик к хорошему плану. Если планировщик по умолчанию выбирает неудачный порядок соединения, вы можете заставить его выбрать лучший, применив синтаксис JOIN, конечно если вы сами его знаете. Эффект подобной оптимизации рекомендуется подтверждать экспериментально.

На время планирования влияет и другой, тесно связанный фактор — решение о включении подзапросов в родительский запрос. Пример такого запроса:

```
SELECT *
FROM x, y,
    (SELECT * FROM a, b, c WHERE something) AS ss
WHERE somethingelse;
```

Такая же ситуация может возникнуть с представлением, содержащим соединение; вместо ссылки на это представление будет вставлено его выражение SELECT и в результате получится запрос, похожий на показанный выше. Обычно планировщик старается включить подзапрос в родительский запрос и получить таким образом:

```
SELECT * FROM x, y, a, b, c WHERE something AND somethingelse;
```

Часто это позволяет построить лучший план, чем при планировании подзапросов по отдельности. (Например, внешние условия WHERE могут быть таковы, что при соединении сначала X с A будет исключено множество строк A, а значит формировать логический результат подзапроса полностью не потребуется.) Но в то же время тем самым мы увеличиваем время планирования; две задачи соединения трёх элементов мы заменяем одной с пятью элементами. Так как число вариантов увеличивается экспоненциально, сложность задачи увеличивается многократно. Планировщик пытается избежать проблем поиска с огромным числом вариантов, рассматривая подзапросы отдельно, если в предложении FROM родительского запроса оказывается больше чем from_collapse_limit элементов. Изменяя этот параметр выполнения, можно подобрать оптимальное соотношение времени планирования и качества плана.

Параметры from_collapse_limit и join_collapse_limit называются похоже, потому что они делают практически одно и то же: первый параметр определяет, когда планировщик будет «сносить» в предложение FROM подзапросы, а второй — явные соединения. Обычно join_collapse_limit устанавливается равным from_collapse_limit (чтобы явные соединения и подзапросы обрабатывались одинаково) или 1 (если требуется управлять порядком соединений). Но вы можете задать другие значения, чтобы добиться оптимального соотношения времени планирования и времени выполнения запросов.

14.4. Наполнение базы данных

Довольно часто в начале или в процессе использования базы данных возникает необходимость загрузить в неё большой объём данных. В этом разделе приведены рекомендации, которые помогут сделать это максимально эффективно.

14.4.1. Отключите автофиксацию транзакций

Выполняя серию команд INSERT, выключите автофиксацию транзакций и зафиксируйте транзакцию только один раз в самом конце. (В обычном SQL это означает, что нужно выполнить BEGIN до, и COMMIT после этой серии. Некоторые клиентские библиотеки могут делать это автоматически, в таких случаях нужно убедиться, что это так.) Если вы будете фиксировать каждое добавление по отдельности, Postgres Pro придётся проделать много действий для каждой добавляемой строки. Выполнять все операции в одной транзакции хорошо ещё и потому, что в случае ошибки добавления одной из строк произойдёт откат к исходному состоянию и вы не окажетесь в сложной ситуации с частично загруженными данными.

14.4.2. Используйте COPY

Используйте COPY, чтобы загрузить все строки одной командой вместо серии INSERT. Команда COPY оптимизирована для загрузки большого количества строк; хотя она не так гибка, как INSERT, но при загрузке больших объёмов данных она влечёт гораздо меньше накладных расходов. Так как COPY — это одна команда, применяя её, нет необходимости отключать автофиксацию транзакций.

В случаях, когда COPY не подходит, может быть полезно создать подготовленный оператор INSERT с помощью PREPARE, а затем выполнять EXECUTE столько раз, сколько потребуется. Это позволит избежать накладных расходов, связанных с разбором и анализом каждой команды INSERT. В разных интерфейсах это может выглядеть по-разному; за подробностями обратитесь к описанию «подготовленных операторов» в документации конкретного интерфейса.

Заметьте, что с помощью COPY большое количество строк практически всегда загружается быстрее, чем с помощью INSERT, даже если используется PREPARE и серия операций добавления заключена в одну транзакцию.

COPY работает быстрее всего, если она выполняется в одной транзакции с командами CREATE TABLE или TRUNCATE. В таких случаях записывать WAL не нужно, так как в случае ошибки файлы, содержащие загружаемые данные, будут всё равно удалены. Однако это замечание справедливо только для несекционированных таблиц, когда параметр wal_level равен minimal, так как в противном случае все команды должны записывать свои изменения в WAL.

14.4.3. Удалите индексы

Если вы загружаете данные в только что созданную таблицу, быстрее всего будет загрузить данные с помощью COPY, а затем создать все необходимые для неё индексы. На создание индекса для уже существующих данных уйдёт меньше времени, чем на последовательное его обновление при добавлении каждой строки.

Если вы добавляете данные в существующую таблицу, может иметь смысл удалить индексы, загрузить таблицу, а затем пересоздать индексы. Конечно, при этом надо учитывать, что временное отсутствие индексов может отрицательно повлиять на скорость работы других пользователей. Кроме того, следует дважды подумать, прежде чем удалять уникальные индексы, так как без них соответствующие проверки ключей не будут выполняться.

14.4.4. Удалите ограничения внешних ключей

Как и с индексами, проверки, связанные с ограничениями внешних ключей, выгоднее выполнять «массово», а не для каждой строки в отдельности. Поэтому может быть полезно удалить ограничения внешних ключей, загрузить данные, а затем восстановить прежние ограничения. И в этом случае тоже приходится выбирать между скоростью загрузки данных и риском допустить ошибки в отсутствие ограничений.

Более того, когда вы загружаете данные в таблицу с существующими ограничениями внешнего ключа, для каждой новой строки добавляется запись в очередь событий триггера (так как именно срабатывающий триггер проверяет такие ограничения для строки). При загрузке многих миллионов строк очередь событий триггера может занять всю доступную память, что приведёт к недопустимой нагрузке на файл подкачки или даже к сбою команды. Таким образом, загружая большие объёмы данных, может быть не просто желательно, а *необходимо* удалять, а затем восстанавливать внешние ключи. Если же временное отключение этого ограничения неприемлемо, единственно возможным решением может быть разделение всей операции загрузки на меньшие транзакции.

14.4.5. Увеличьте maintenance_work_mem

Ускорить загрузку больших объёмов данных можно, увеличив параметр конфигурации maintenance_work_mem на время загрузки. Это приведёт к увеличению быстродействия CREATE INDEX и ALTER TABLE ADD FOREIGN KEY. На скорость самой команды COPY это не повлияет, так что этот совет будет полезен, только если вы применяете какой-либо из двух вышеописанных приёмов.

14.4.6. Увеличьте max_wal_size

Также массовую загрузку данных можно ускорить, изменив на время загрузки параметр конфигурации max_wal_size. Загружая большие объёмы данных, Postgres Pro вынужден увеличивать частоту контрольных точек по сравнению с обычной (которая задаётся параметром checkpoint_timeout), а значит и чаще сбрасывать «грязные» страницы на диск. Временно увеличив max_wal_size, можно уменьшить частоту контрольных точек и связанных с ними операций ввода-вывода.

14.4.7. Отключите архивацию WAL и потоковую репликацию

Для загрузки больших объёмов данных в среде, где используется архивация WAL или потоковая репликация, быстрее будет сделать копию базы данных после загрузки данных, чем обрабатывать множество операций изменений в WAL. Чтобы отключить передачу изменений через WAL в процессе загрузки, отключите архивацию и потоковую репликацию, назначьте параметру wal_level значение minimal, archive_mode — off, а max_wal_senders — 0. Но имейте в виду, что изменённые параметры вступят в силу только после перезапуска сервера.

Это не только поможет сэкономить время архивации и передачи WAL, но и непосредственно ускорит некоторые команды, которые могут вовсе не использовать WAL, если wal_level равен minimal. (Они могут гарантировать безопасность при сбое, не записывая все изменения в WAL, а выполнив только fsync в конце операции, что будет гораздо дешевле.) Это относится к следующим командам:

- CREATE TABLE AS SELECT
- CREATE INDEX (и подобные команды, как например ALTER TABLE ADD PRIMARY KEY)
- ALTER TABLE SET TABLESPACE
- CLUSTER
- COPY FROM, когда целевая таблица была создана или опустошена ранее в той же транзакции

14.4.8. Выполните в конце ANALYZE

Всякий раз, когда распределение данных в таблице значительно меняется, настоятельно рекомендуется выполнять ANALYZE. Эта рекомендация касается и загрузки в таблицу большого объёма данных. Выполнив ANALYZE (или VACUUM ANALYZE), вы тем самым обновите статистику по данной таблице для планировщика. Когда планировщик не имеет статистики или она не соответствует действительности, он не сможет правильно планировать запросы, что приведёт к снижению быстродействия при работе с соответствующими таблицами. Заметьте, что если включён демон автоочистки, он может запускать ANALYZE автоматически; подробнее об этом можно узнать в Подразделе 23.1.3 и Подразделе 23.1.6.

14.4.9. Несколько замечаний относительно pg_dump

В скриптах загрузки данных, которые генерирует pg_dump, автоматически учитываются некоторые, но не все из этих рекомендаций. Чтобы загрузить данные, которые выгрузил pg_dump, максимально быстро, вам нужно будет выполнить некоторые дополнительные действия вручную. (Заметьте, что эти замечания относятся только к *восстановлению* данных, но не к *выгрузке* их. Следующие рекомендации применимы вне зависимости от того, загружается ли архивный файл pg_dump в psql или в pg_restore.)

По умолчанию pg_dump использует команду COPY и когда она выгружает полностью схему и данные, в сгенерированном скрипте она сначала предусмотрительно загружает данные, а потом создаёт индексы и внешние ключи. Так что в этом случае часть рекомендаций выполняется автоматически. Вам остаётся учесть только следующие:

- Установите подходящие (то есть превышающие обычные) значения для maintenance_work_mem и max_wal_size.

- Если вы используете архивацию WAL или потоковую репликацию, по возможности отключите их на время восстановления. Для этого перед загрузкой данных, присвойте параметру `archive_mode` значение `off`, `wal_level` — `minimal`, а `max_wal_senders` — 0. Закончив восстановление, верните их обычные значения и сделайте свежую базовую резервную копию.
- Поэкспериментируйте с режимами параллельного копирования и восстановления команд pg_dump и pg_restore, и подберите оптимальное число параллельных заданий. Параллельное копирование и восстановление данных, управляемое параметром `-j`, должно дать значительный выигрыш в скорости по сравнению с последовательным режимом.
- Если это возможно в вашей ситуации, восстановите все данные в рамках одной транзакции. Для этого передайте параметр `-1` или `--single-transaction` команде psql или pg_restore. Но учтите, что в этом режиме даже незначительная ошибка приведёт к откату всех изменений и часы восстановления будут потрачены зря. В зависимости от того, насколько взаимосвязаны данные, предпочтительнее может быть вычистить их вручную. Команды `COPY` будут работать максимально быстро, когда они выполняются в одной транзакции и архивация WAL выключена.
- Если на сервере баз данных установлено несколько процессоров, полезным может оказаться параметр `--jobs` команды pg_restore. С его помощью можно выполнить загрузку данных и создание индексов параллельно.
- После загрузки данных запустите `ANALYZE`.

При выгрузке данных без схемы тоже используется команда `COPY`, но индексы, как обычно и внешние ключи, при этом не удаляются и не пересоздаются. [1] Поэтому, загружая только данные, вы сами должны решить, нужно ли для ускорения загрузки удалять и пересоздавать индексы и внешние ключи. При этом будет так же полезно увеличить параметр `max_wal_size`, но не `maintenance_work_mem`; его стоит менять, только если вы впоследствии пересоздаёте индексы и внешние ключи вручную. И не забудьте выполнить `ANALYZE` после; подробнее об этом можно узнать в Подразделе 23.1.3 и Подразделе 23.1.6.

14.5. Оптимизация, угрожающая стабильности

Стабильность — это свойство базы данных, гарантирующее, что результат зафиксированных транзакций будет сохранён даже в случае сбоя сервера или отключения питания. Однако обеспечивается стабильность за счёт значительной дополнительной нагрузки. Поэтому, если вы можете отказаться от такой гарантии, Postgres Pro можно ускорить ещё больше, применив следующие методы оптимизации. Кроме явно описанных исключений, даже с такими изменениями конфигурации при сбое программного ядра СУБД гарантия стабильности сохраняется; риск потери или разрушения данных возможен только в случае внезапной остановки операционной системы.

- Поместите каталог данных кластера БД в файловую систему, размещённую в памяти (т. е. в RAM-диск). Так вы исключите всю активность ввода/вывода, связанную с базой данных, если только размер базы данных не превышает объём свободной памяти (возможно, с учётом файла подкачки).
- Выключите fsync; сбрасывать данные на диск не нужно.
- Выключите synchronous_commit; нет необходимости принудительно записывать WAL на диск при фиксации каждой транзакции. Но учтите, это может привести к потере транзакций (хотя данные останутся согласованными) в случае сбоя *базы данных*.
- Выключите full_page_writes; защита от частичной записи страниц не нужна.
- Увеличьте max_wal_size и checkpoint_timeout; это уменьшит частоту контрольных точек, хотя объём `/pg_wal` при этом вырастет.
- Создавайте нежурналируемые таблицы для оптимизации записи в WAL (но учтите, что такие таблицы не защищены от сбоя).

[1] Вы можете отключить внешние ключи, используя параметр `--disable-triggers` — но при этом нужно понимать, что тем самым вы не просто отложите, а полностью выключите соответствующие проверки, что позволит вставить недопустимые данные.

Глава 15. Параллельный запрос

Postgres Pro может вырабатывать такие планы запросов, которые будут задействовать несколько CPU, чтобы получить ответ на запросы быстрее. Эта возможность называется распараллеливанием запросов. Для многих запросов параллельное выполнение не даёт никакого выигрыша, либо из-за ограничений текущей реализации, либо из-за принципиальной невозможности построить параллельный план, который был бы быстрее последовательного. Однако для запросов, в которых это может быть полезно, распараллеливание часто даёт очень значительное ускорение. Многие такие запросы могут выполняться в параллельном режиме как минимум двое быстрее, а некоторые — быстрее в четыре и даже более раз. Обычно наибольший выигрыш можно получить с запросами, обрабатывающими большой объём данных, но возвращающими пользователю всего несколько строк. В этой главе достаточно подробно рассказывается, как работают параллельные запросы и в каких ситуациях их можно использовать, чтобы пользователи, желающие применять их, понимали, чего ожидать.

15.1. Как работают параллельно выполняемые запросы

Когда оптимизатор определяет, что параллельное выполнение будет наилучшей стратегией для конкретного запроса, он создаёт план запроса, включающий *узел Gather* (Сбор) или *Gather Merge* (Сбор со слиянием). Взгляните на простой пример:

```
EXPLAIN SELECT * FROM pgbench_accounts WHERE filler LIKE '%x%';
                                     QUERY PLAN
------------------------------------------------------------------------------------
 Gather  (cost=1000.00..217018.43 rows=1 width=97)
   Workers Planned: 2
   ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..216018.33 rows=1 width=97)
         Filter: (filler ~~ '%x%'::text)
(4 rows)
```

Во всех случаях узел `Gather` или `Gather Merge` будет иметь ровно один дочерний план, представляющий часть общего плана, выполняемую в параллельном режиме. Если узел `Gather` или `Gather Merge` располагается на самом верху дерева плана, в параллельном режиме будет выполняться весь запрос. Если он находится где-то в другом месте плана, параллельно будет выполняться только часть плана ниже него. В приведённом выше примере запрос обращается только к одной таблице, так что помимо узла `Gather` есть только ещё один узел плана; и так как этот узел является потомком узла `Gather`, он будет выполняться в параллельном режиме.

Используя EXPLAIN, вы можете узнать количество исполнителей, выбранное планировщиком для данного запроса. Когда при выполнении запроса достигается узел `Gather`, процесс, обслуживающий сеанс пользователя, запрашивает фоновые рабочие процессы в этом количестве. Количество исполнителей, которое может попытаться задействовать планировщик, ограничивается значением max_parallel_workers_per_gather. Общее число фоновых рабочих процессов, которые могут существовать одновременно, ограничивается параметрами max_worker_processes и max_parallel_workers. Таким образом, вполне возможно, что параллельный запрос будет выполняться меньшим числом рабочих процессов, чем планировалось, либо вообще без дополнительных рабочих процессов. Оптимальность плана может зависеть от числа доступных рабочих процессов, так что их нехватка может повлечь значительное снижение производительности. Если это наблюдается часто, имеет смысл увеличить `max_worker_processes` и `max_parallel_workers`, чтобы одновременно могло работать больше процессов, либо наоборот уменьшить `max_parallel_workers_per_gather`, чтобы планировщик запрашивал их в меньшем количестве.

Каждый фоновый рабочий процесс, успешно запущенный для данного параллельного запроса, будет выполнять параллельную часть плана. Ведущий процесс также будет выполнять эту часть плана, но он несёт дополнительную ответственность: он должен также прочитать все кортежи, выданные рабочими процессами. Когда параллельная часть плана выдаёт лишь небольшое

количество кортежей, ведущий часто ведёт себя просто как один из рабочих процессов, ускоряя выполнение запроса. И напротив, когда параллельная часть плана выдаёт множество кортежей, ведущий может быть почти всё время занят чтением кортежей, выдаваемых другими рабочими процессами, и выполнять другие шаги обработки, связанные с узлами плана выше узла `Gather` или `Gather Merge`. В таких случаях ведущий процесс может вносить лишь минимальный вклад в выполнение параллельной части плана.

Когда над параллельной частью плана оказывается узел `Gather Merge`, а не `Gather`, это означает, что все процессы, выполняющие части параллельного плана, выдают кортежи в отсортированном порядке, и что ведущий процесс выполняет слияние с сохранением порядка. Узел же `Gather`, напротив, получает кортежи от подчинённых процессов в произвольном удобном ему порядке, нарушая порядок сортировки, который мог существовать.

15.2. Когда может применяться распараллеливание запросов?

Планировщик запросов может отказаться от построения параллельных планов запросов в любом случае под влиянием нескольких параметров. Чтобы он строил параллельные планы запросов при каких-бы то ни было условиях, описанные далее параметры необходимо настроить указанным образом.

- max_parallel_workers_per_gather должен иметь значение, большее нуля. Это особый вариант более общего ограничения на суммарное число используемых рабочих процессов, задаваемого параметром `max_parallel_workers_per_gather`.
- dynamic_shared_memory_type должен иметь значение, отличное от `none`. Для параллельного выполнения запросов нужна динамическая общая память, через которую будут передаваться данные между взаимодействующими процессами.

В дополнение к этому, система должна работать не в однопользовательском режиме. Так как в этом режиме вся СУБД работает в одном процессе, фоновые рабочие процессы в нём недоступны.

Даже если принципиально возможно построить параллельные планы выполнения, планировщик не будет строить такой план для определённого запроса, если имеет место одно из следующих обстоятельств:

- Запрос выполняет запись данных или блокирует строки в базе данных. Если запрос содержит операцию, изменяющую данные либо на верхнем уровне, либо внутри CTE, для такого запроса не будут строиться параллельные планы. Это ограничение текущей реализации, которое может быть смягчено в будущих версиях.
- Запрос может быть приостановлен в процессе выполнения. В ситуациях, когда система решает, что может иметь место частичное или дополнительное выполнение, план параллельного выполнения не строится. Например, курсор, созданный предложением DECLARE CURSOR, никогда не будет использовать параллельный план. Подобным образом, цикл PL/pgSQL вида `FOR x IN query LOOP .. END LOOP` никогда не будет использовать параллельный план, так как система параллельных запросов не сможет определить, может ли безопасно выполняться код внутри цикла во время параллельного выполнения запроса.
- В запросе используются функции, помеченные как `PARALLEL UNSAFE`. Большинство системных функций безопасны для параллельного выполнения (`PARALLEL SAFE`), но пользовательские функции по умолчанию помечаются как небезопасные (`PARALLEL UNSAFE`). Эта характеристика функции рассматривается в Разделе 15.4.
- Запрос работает внутри другого запроса, уже параллельного. Например, если функция, вызываемая в параллельном запросе, сама выполняет SQL-запрос, последний запрос никогда не будет выполняться параллельно. Это ограничение текущей реализации, но убирать его вряд ли следует, так как это может привести к использованию одним запросом чрезмерного количества процессов.
- Для транзакции установлен сериализуемый уровень изоляции. Это ограничение текущей реализации.

Даже когда для определённого запроса построен параллельный план, возможны различные обстоятельства, при которых этот план нельзя будет выполнить в параллельном режиме. В этих случаях ведущий процесс выполнит часть плана ниже узла `Gather` полностью самостоятельно, как если бы узла `Gather` вовсе не было. Это произойдёт только при выполнении одного из следующих условий:

- Невозможно получить ни одного фонового рабочего процесса из-за ограничения общего числа этих процессов значением max_worker_processes.
- Невозможно получить ни одного фонового рабочего процесса из-за ограничения общего числа таких процессов для параллельного выполнения значением max_parallel_workers.
- Клиент передаёт сообщение Execute с ненулевым количеством выбираемых кортежей. За подробностями обратитесь к описанию протокола расширенных запросов. Так как libpq в настоящее время не позволяет передавать такие сообщения, это возможно только с клиентом, задействующим не libpq. Если это происходит часто, имеет смысл установить в max_parallel_workers_per_gather 0 в сеансах, для которых это актуально, чтобы система не пыталась строить планы, которые могут быть неэффективны при последовательном выполнении.
- Подготовленный оператор выполняется в конструкции `CREATE TABLE .. AS EXECUTE ...`. Эта конструкция меняет характер операции с «только чтение» на «чтение+запись», что исключает параллельное выполнение этой операции.
- Для транзакции установлен сериализуемый уровень изоляции. Обычно эта ситуация не возникает, так как при таком уровне изоляции не строятся параллельные планы выполнения. Однако она возможна, если уровень изоляции транзакции меняется на сериализуемый после построения плана и до его выполнения.

15.3. Параллельные планы

Так как каждый рабочий процесс выполняет параллельную часть плана до конца, нельзя просто взять обычный план запроса и запустить его в нескольких исполнителях. В этом случае все исполнители выдавали бы полные копии выходного набора результатов, так что запрос выполнится не быстрее, чем обычно, а его результаты могут быть некорректными. Вместо этого параллельной частью плана должно быть то, что для оптимизатора представляется как *частичный план*; то есть такой план, при выполнении которого в отдельном процессе будет получено только подмножество выходных строк, а каждая требующаяся строка результата будет гарантированно выдана ровно одним из сотрудничающих процессов. Вообще говоря, это означает, что сканирование нижележащей таблицы запроса должно проводиться с учётом распараллеливания.

15.3.1. Параллельные сканирования

В настоящее время поддерживаются следующие виды сканирований таблицы, рассчитанные на параллельное выполнение.

- При *параллельном последовательном сканировании* блоки таблицы будут разделены между взаимодействующими процессами. Блоки выдаются по очереди, так что доступ к таблице остаётся последовательным.
- При *параллельном сканировании кучи по битовой карте* один процесс выбирается на роль ведущего. Этот процесс производит сканирование одного или нескольких индексов и строит битовую карту, показывающую, какие блоки таблицы нужно посетить. Затем эти блоки разделяются между взаимодействующими процессами как при параллельном последовательном сканировании. Другими словами, сканирование кучи выполняется в параллельном режиме, а сканирование нижележащего индекса — нет.
- При *параллельном сканировании по индексу* или *параллельном сканировании только индекса* взаимодействующие процессы читают данные из индекса по очереди. В настоящее время параллельное сканирование индекса поддерживается только для индексов-B-деревьев. Каждый процесс будет выбирать один блок индекса с тем, чтобы просканировать и вернуть все кортежи, на которые он ссылается; другие процессы могут в то же время возвращать

кортежи для другого блока индекса. Результаты параллельного сканирования B-дерева каждый рабочий процесс возвращает в отсортированном порядке.

В будущем может появиться поддержка параллельного выполнения и для других вариантов сканирования, например, сканирования индексов, отличных от B-дерева.

15.3.2. Параллельные соединения

Как и в непараллельном плане, целевая таблица может соединяться с одной или несколькими другими таблицами с использованием вложенных циклов, соединений по хешу или соединений слиянием. Внутренней стороной соединения может быть любой вид непараллельного плана, который в остальном поддерживается планировщиком, при условии, что он безопасен для выполнения в параллельном исполнителе. Например, если выбрано соединение с вложенным циклом, во внутреннем плане может быть сканирование индекса, при котором находится значение, взятое из внешней стороны соединения.

Каждый рабочий процесс будет выполнять внутреннюю сторону соединения в полном объёме. Обычно это не проблема для вложенных циклов, но это может быть неэффективно с соединением по хешу или соединением слиянием. Например, для соединения по хешу это ограничение означает, что в каждом рабочем процессе будет строиться одна и та же хеш-таблица, что приемлемо для маленьких таблиц, но может быть неэффективно, когда внутренняя таблица велика. Для соединения слиянием это может означать, что каждый рабочий процесс независимо выполняет сортировку внутреннего отношения, потенциально медленную операцию. Конечно, в тех случаях, когда параллельный план такого вида может быть неэффективным, планировщик запросов обычно выбирает другой план (возможно, уже без распараллеливания).

15.3.3. Параллельное агрегирование

Postgres Pro поддерживает параллельное агрегирование, выполняя агрегирование в два этапа. Сначала каждый процесс, задействованный в параллельной части запроса, выполняет шаг агрегирования, выдавая частичный результат для каждой известной ему группы. В плане это отражает узел `Partial Aggregate`. Затем эти промежуточные результаты передаются ведущему через узел `Gather` или `Gather Merge`. И наконец, ведущий заново агрегирует результаты всех рабочих процессов, чтобы получить окончательный результат. Это отражает в плане узел `Finalize Aggregate`.

Так как узел `Finalize Aggregate` выполняется в ведущем процессе, запросы, выдающие достаточно большое количество групп по отношению к числу входных строк, будут расцениваться планировщиком как менее предпочтительные. Например, в худшем случае количество групп, выявленных узлом `Finalize Aggregate`, может равняться числу входных строк, обработанных всеми рабочими процессами на этапе `Partial Aggregate`. Очевидно, что в такой ситуации использование параллельного агрегирования не даст никакого выигрыша производительности. Планировщик запросов учитывает это в процессе планирования, так что выбор параллельного агрегирования в подобных случаях очень маловероятен.

Параллельное агрегирование поддерживается не во всех случаях. Чтобы оно поддерживалось, агрегатная функция должна быть безопасной для распараллеливания и должна иметь комбинирующую функцию. Если переходное состояние агрегатной функции имеет тип `internal`, она должна также иметь функции сериализации и десериализации. За подробностями обратитесь к CREATE AGGREGATE. Параллельное агрегирование не поддерживается, если вызов агрегатной функции содержит предложение `DISTINCT` или `ORDER BY`. Также оно не поддерживается для сортирующих агрегатов или когда запрос включает предложение `GROUPING SETS`. Оно может использоваться только когда все соединения, задействованные в запросе, также входят в параллельную часть плана.

15.3.4. Советы по параллельным планам

Если для запроса ожидается параллельный план, но такой план не строится, можно попытаться уменьшить parallel_setup_cost или parallel_tuple_cost. Разумеется, этот план может оказаться

медленнее последовательного плана, предпочитаемого планировщиком, но не всегда. Если вы не получаете параллельный план даже с очень маленькими значениями этих параметров (например, сбросив оба их в ноль), может быть какая-то веская причина тому, что планировщик запросов не может построить параллельный план для вашего запроса. За информацией о возможных причинах обратитесь к Разделу 15.2 и Разделу 15.4.

Когда выполняется параллельный план, вы можете применить `EXPLAIN (ANALYZE, VERBOSE)`, чтобы просмотреть статистику по каждому узлу плана в разрезе рабочих процессов. Это может помочь определить, равномерно ли распределяется работа между всеми узлами плана, и на более общем уровне понимать характеристики производительности плана.

15.4. Безопасность распараллеливания

Планировщик классифицирует операции, вовлечённые в выполнение запроса, как либо *безопасные для распараллеливания*, либо *ограниченно распараллеливаемые*, либо *небезопасные для распараллеливания*. Безопасной для распараллеливания операцией считается такая, которая не мешает параллельному выполнению запроса. Ограниченно распараллеливаемой операцией считается такая, которая не может выполняться в параллельном рабочем процессе, но может выполняться в ведущем процессе, когда запрос выполняется параллельно. Таким образом, ограниченно параллельные операции никогда не могут оказаться ниже узла `Gather` или `Gather Merge`, но могут встречаться в других местах плана, содержащего такой узел. Небезопасные для распараллеливания операции не могут выполняться в параллельных запросах, даже в ведущем процессе. Когда запрос содержит что-либо небезопасное для распараллеливания, параллельное выполнение для такого запроса полностью исключается.

Ограниченно распараллеливаемыми всегда считаются следующие операции:

- Сканирование общих табличных выражений (CTE).
- Сканирование временных таблиц.
- Сканирование сторонних таблиц, если только обёртка сторонних данных не предоставляет функцию `IsForeignScanParallelSafe`, которая допускает распараллеливание.
- Доступ к `InitPlan` или связанному `SubPlan`.

15.4.1. Пометки параллельности для функций и агрегатов

Планировщик не может автоматически определить, является ли пользовательская обычная или агрегатная функция безопасной для распараллеливания, так как это потребовало бы предсказания действия каждой операции, которую могла бы выполнять функция. В общем случае это равнозначно решению проблемы остановки, а значит, невозможно. Даже для простых функций, где это в принципе возможно, мы не пытаемся это делать, так как это будет слишком дорогой и потенциально неточной процедурой. Вместо этого, все определяемые пользователем функции полагаются небезопасными для распараллеливания, если явно не отмечено обратное. Когда используется CREATE FUNCTION или ALTER FUNCTION, функции можно назначить отметку `PARALLEL SAFE`, `PARALLEL RESTRICTED` или `PARALLEL UNSAFE`, отражающую её характер. В команде CREATE AGGREGATE для параметра `PARALLEL` можно задать `SAFE`, `RESTRICTED` или `UNSAFE` в виде соответствующего значения.

Обычные и агрегатные функции должны помечаться небезопасными для распараллеливания (`PARALLEL UNSAFE`), если они пишут в базу данных, обращаются к последовательностям, изменяют состояние транзакции, даже временно (как, например, функция PL/pgSQL, устанавливающая блок `EXCEPTION` для перехвата ошибок), либо производят постоянные изменения параметров. Подобным образом, функции должны помечаться как ограниченно распараллеливаемые (`PARALLEL RESTRICTED`), если они обращаются к временным таблицам, состоянию клиентского подключения, курсорам, подготовленным операторам или разнообразному локальному состоянию обслуживающего процесса, которое система не может синхронизировать между рабочими процессами. Например, по этой причине ограниченно параллельными являются функции `setseed` и `random`.

В целом, если функция помечена как безопасная, когда на самом деле она небезопасна или ограниченно безопасна, или если она помечена как ограниченно безопасная, когда на самом деле она небезопасная, такая функция может выдавать ошибки или возвращать неправильные ответы при использовании в параллельном запросе. Функции на языке C могут теоретически проявлять полностью неопределённое появление при некорректной пометке, так как система никаким образом не может защитить себя от произвольного кода C, но чаще всего результат будет не хуже, чем с любой другой функцией. В случае сомнений, вероятно, лучше всего будет помечать функции как небезопасные (`UNSAFE`).

Если функция, выполняемая в параллельном рабочем процессе, затребует блокировки, которыми не владеет ведущий, например, обращаясь к таблице, не упомянутой в запросе, эти блокировки будут освобождены по завершении процесса, а не в конце транзакции. Если вы разрабатываете функцию с таким поведением, и эта особенность выполнения оказывается критичной, пометьте такую функцию как `PARALLEL RESTRICTED`, чтобы она выполнялась только в ведущем процессе.

Заметьте, что планировщик запросов не рассматривает возможность отложенного выполнения ограниченно распараллеливаемых обычных или агрегатных функций, задействованных в запросе, для получения лучшего плана. Поэтому, например, если предложение `WHERE`, применяемое к конкретной таблице, является ограниченно параллельным, планировщик запросов исключит возможность сканирования этой таблицы в параллельной части плана. В некоторых случаях возможно (и, вероятно, более эффективно) включить сканирование этой таблицы в параллельную часть запроса и отложить вычисление предложения `WHERE`, чтобы оно происходило над узлом `Gather`, но планировщик этого не делает.

Часть III. Администрирование сервера

В этой части документации освещаются темы, представляющие интерес для администратора баз данных Postgres Pro. В частности, здесь рассматривается установка программного обеспечения, установка и настройка сервера, управление пользователями и базами данных, а также задачи обслуживания. С этими темами следует ознакомиться всем, кто эксплуатирует сервер Postgres Pro (даже для личных целей, а тем более в производственной среде).

Материал этой части даётся примерно в том порядке, в каком его следует читать начинающему пользователю. При этом её главы самодостаточны и при желании могут быть прочитаны по отдельности. Информация в этой части книги представлена в повествовательном стиле и разделена по темам. Если же вас интересует формальное и полное описание определённой команды, см. Часть VI.

Первые несколько глав написаны так, чтобы их можно было понять без предварительных знаний, так что начинающие пользователи, которым нужно установить свой собственный сервер, могут начать свой путь с них. Остальные главы части посвящены настройке сервера и управлению им; в этом материале подразумевается, что читатель знаком с основными принципами использования СУБД Postgres Pro. За дополнительной информацией мы рекомендуем читателям обратиться к Части I и Части II.

Глава 16. Установка из двоичных пакетов

16.1. Установка Postgres Pro Standard в Linux

Для систем на базе Linux Postgres Pro Standard поставляется в виде двоичных пакетов. Все двоичные дистрибутивы Postgres Pro состоят из нескольких пакетов. Структура пакетов отличается от структуры ванильного PostgreSQL и поддерживает следующие режимы установки:

- **Быстрая установка и настройка**. Пакет `postgrespro-std-10` устанавливает и настраивает все компоненты, необходимые для получения готовой к использованию конфигурации, как клиентских, так и серверных компонентов. Выберите этот вариант, если вы планируете установить только один экземпляр Postgres Pro и вас не беспокоят возможные конфликты с другими продуктами на базе PostgreSQL.

  > **Важно**
  >
  > При установке пакета `postgrespro-std-10` могут быть удалены существующие инсталляции продуктов на базе Postgres Pro и PostgreSQL. И напротив, установленный пакет Postgres Pro может быть автоматически удалён, если вы впоследствии развернёте другой продукт на базе PostgreSQL. Данный пакет не нужно устанавливать в процессе обновления или миграции, а также если вы планируете разворачивать в одной системе несколько инсталляций.

- **Расширенная установка**. Вы можете выбрать любые пакеты, требующиеся для ваших целей, включая пакеты для разработчиков. Этот вариант требует ручной настройки, поэтому для него необходимо хорошо разбираться в Linux и понимать архитектуру PostgreSQL. Это единственно возможный вариант, если вы планируете реализовать один из следующих сценариев использования Postgres Pro:
  - Установка одновременно нескольких версий Postgres Pro либо установка вместе с другими продуктами на базе PostgreSQL.
  - Обновление с другой версии или миграция с другого продукта на базе PostgreSQL.
  - Управление сервером Postgres Pro с использованием программных средств обеспечения высокой степени доступности, таких как pacemaker, вместо стандартных системных механизмов управления службами.

Минимальные системные требования для установки Postgres Pro, создания кластера баз данных и запуска сервера:

- Процессор с частотой 1 ГГц
- 1 ГБ ОЗУ
- 1 ГБ свободного дискового пространства

Для вспомогательных компонентов или данных потребуется дополнительное дисковое пространство.

16.1.1. Поддерживаемые дистрибутивы Linux

Двоичные пакеты Postgres Pro предоставляются для следующих систем на базе Linux:

- Системы Red Hat Enterprise Linux (RHEL) и производные от них: CentOS 7/8, Red Hat Enterprise Linux 7/8, Oracle Linux 7/8, Rosa Enterprise Linux Server 7, РОСА «КОБАЛЬТ» (серверная редакция) на платформе РОСА 7, РЕД ОС МУРОМ 7.2, AlterOS 7.5
- Системы на базе Debian: Debian 9/10, Ubuntu 18.04/20.04, Astra Linux «Смоленск» 1.6/1.7, Astra Linux «Орёл» 2.12
- Альт 8/9/10, Альт Линукс СПТ 7.0, Альт 8 СП, Альт 8.2 СП

- SUSE Linux Enterprise Server (SLES) 12/15

> **Примечание**
>
> Двоичные пакеты Postgres Pro используют библиотеку tzdata, предоставляемую операционной системой, поэтому важно, чтобы в системе была установлена последняя её версия. В случае использования устаревшей библиотеки tzdata значения времени в вашей базе данных могут быть некорректными.

16.1.2. Быстрая установка и настройка

Если вам нужно установить только один экземпляр Postgres Pro и вы не собираетесь использовать никакие другие продукты на базе PostgreSQL в вашей системе, вы можете использовать режим быстрой установки. Типичная процедура установки в этом случае выглядит так:

1. Подключите репозиторий пакетов, предназначенный для вашей операционной системы. Конкретные адреса репозиториев и команды для их подключения в поддерживаемых дистрибутивах Linux вы можете найти на *Странице загрузки* для соответствующей версии Postgres Pro.

2. Установите пакет `postgrespro-std-10`. При этом по зависимостям установятся все требуемые компоненты, будет создана база данных по умолчанию, запущен сервер баз данных и настроен автозапуск сервера при загрузке системы, а все предоставляемые программы станут доступными в пути `PATH`.

> **Примечание**
>
> По умолчанию конфигурация базы данных настраивается под устанавливаемый продукт Postgres Pro. Если вам нужна другая настройка сервера базы данных, выберите расширенную установку и воспользуйтесь аргументом `tune` команды `pg-setup initdb`.

После завершения установки вы можете запустить psql от имени пользователя `postgres` и подключиться к только что созданной базе данных, находящейся в каталоге данных `/var/lib/pgpro/std-10/data`.

Так как база данных по умолчанию создаётся скриптом pg-setup, путь к каталогу данных сохраняется в файле `/etc/default/postgrespro-std-10`. Все последующие команды pg-setup, а также любые команды, управляющие службой Postgres Pro, будут нацелены именно на эту базу данных. Кроме того, в указанном файле можно изменить значение `PG_OOM_ADJUST_VALUE` для дочерних процессов postmaster (за подробностями обратитесь к Подразделу 17.4.4).

16.1.3. Расширенная установка

Разделение дистрибутива на несколько пакетов позволяет произвести инсталляцию по-разному для различных применений: для серверов баз данных, клиентских систем или рабочих станций разработчиков. Такие инсталляции необходимо настраивать вручную, но это даёт дополнительную гибкость при использовании продукта. Вы можете установить несколько версий Postgres Pro параллельно, а также вместе с другими продуктами PostgreSQL. В частности, это может потребоваться при осуществлении обновления или при миграции с другого продукта на базе PostgreSQL.

Для осуществления расширенной установки выполните следующие действия:

1. Подключите репозиторий пакетов, предназначенный для вашей операционной системы. Конкретные адреса репозиториев и команды для их подключения в поддерживаемых

дистрибутивах Linux вы можете найти на *Странице загрузки* для соответствующей версии Postgres Pro.

2. Выберите пакеты Postgres Pro, требующиеся для ваших целей, и установите их с помощью стандартных для вашего дистрибутива Linux команд. Доступные пакеты перечислены в Таблице 16.1.

   В результате все файлы будут установлены в каталог `/opt/pgpro/std-10`.

3. Запустите от имени root утилиту pg-wrapper, чтобы добавить и клиентские, и серверные установленные программы в путь поиска `PATH`, а также включить страницы man по SQL в файл конфигурации страниц man. Эта утилита входит в состав пакета `postgrespro-std-10-client`.

   ```
   /opt/pgpro/std-10/bin/pg-wrapper links update
   ```

   Подробнее о разрешении возможных конфликтов рассказывается в pg-wrapper.

4. Если вы устанавливаете пакет `postgrespro-std-10-server`, выполните следующую процедуру настройки сервера:

   a. Создайте начальную базу данных, запустив вспомогательный скрипт pg-setup от имени root с ключом `initdb`:

      ```
      /opt/pgpro/std-10/bin/pg-setup initdb [--tune=конфигурация] [параметры_initdb]
      ```

      Здесь:

      - Аргумент `tune` выбирает вариант конфигурации базы данных.
      - *`параметры_initdb`* — обычные параметры initdb.

      > **Примечание**
>
> По умолчанию pg-setup инициализирует кластер баз данных со включёнными контрольными суммами. Если для вас это нежелательно, укажите `--no-data-checksums`.

      Скрипт `pg-setup` выполняет административные операции с базами данных от имени пользователя `postgres`. Если вы не укажете никакие параметры initdb, база по умолчанию создаётся в каталоге `/var/lib/pgpro/std-10/data` с параметрами локализации, определяемыми переменной окружения `LANG` текущего сеанса. Переменные окружения `LC_*` при этом игнорируются.

      Так как база данных по умолчанию создаётся скриптом pg-setup, путь к каталогу данных сохраняется в файле `/etc/default/postgrespro-std-10`. Все последующие команды pg-setup, а также любые команды, управляющие службой Postgres Pro, будут нацелены именно на эту базу данных. Кроме того, в указанном файле можно изменить значение `PG_OOM_ADJUST_VALUE` для дочерних процессов postmaster (за подробностями обратитесь к Подразделу 17.4.4).

   b. Запустите сервер с помощью `pg-setup` от имени root следующим образом:

      ```
      /opt/pgpro/std-10/bin/pg-setup service start
      ```

      Как и сервер ванильного PostgreSQL, сервер Postgres Pro работает под именем пользователя `postgres`.

      > **Примечание**
>
> По умолчанию автоматический запуск сервера отключён, чтобы вы могли вручную управлять восстановлением базы данных после перезагрузки системы. Однако вы можете настроить и автоматический запуск сервера Postgres Pro. За подробностями обратитесь к Подразделу 16.1.3.2.

16.1.3.1. Выбор устанавливаемых пакетов

В следующей таблице перечислены все имеющиеся пакеты Postgres Pro.

Таблица 16.1. Пакеты Postgres Pro

Пакет	Описание
postgrespro-std-10	Пакет верхнего уровня, устанавливающий и настраивающий Postgres Pro для серверных и клиентских систем. Не используйте этот пакет при обновлении или миграции. **Важно** При установке пакета postgrespro-std-10 могут быть удалены существующие инсталляции продуктов на базе Postgres Pro и PostgreSQL. И напротив, установленный пакет Postgres Pro может быть автоматически удалён, если вы впоследствии развернёте другой продукт на базе PostgreSQL.
postgrespro-std-10-client	Стандартные клиентские приложения, такие как psql и pg_dump.
postgrespro-std-10-libs	Общие библиотеки, требующиеся для развёртывания клиентских приложений, включая libpq; библиотеки времени выполнения для обработчика ECPG.
postgrespro-std-10-server	Сервер Postgres Pro и серверный язык программирования PL/pgSQL.
postgrespro-std-10-contrib	Дополнительные расширения и программы, разворачиваемые на серверах баз данных.
pg-probackup-std-10	Утилита pg_probackup.
postgrespro-std-10-pgprobackup	Пакет pg_probackup для автоматического обновления с Postgres Pro Standard версии ниже 10.7.1.
postgrespro-std-10-devel	Заголовочные файлы и библиотеки для разработки клиентских приложений и серверных расширений. В системах на базе Debian этот пакет называется postgrespro-std-10-dev.
postgrespro-std-10-plperl	Реализация языка Perl для программирования на стороне сервера.
postgrespro-std-10-plpython	Реализация языка Python для программирования на стороне сервера.
postgrespro-std-10-plpython3	Реализация языка Python 3 для программирования на стороне сервера.
postgrespro-std-10-pltcl	Реализация языка Tcl для программирования на стороне сервера.
postgrespro-std-10-docs	Документация на английском языке.
postgrespro-std-10-docs-ru	Документация на русском языке.

Пакет	Описание
postgrespro-std-10-test	Тестовые скрипты для сервера. Этот пакет представлен только для систем на базе RHEL и SUSE.
pgpro-controldata	Утилита pgpro_controldata, показывающая управляющую информацию кластера БД PostgreSQL/Postgres Pro и параметры совместимости кластера и/или сервера.

Кроме этого, имеются отдельные пакеты с несколькими дополнительными модулями, собранные специально для совместимости с Postgres Pro:

Пакет	Описание
libzstd	Библиотека для быстрого сжатия данных без потерь.
mamonsu	Агент мониторинга для сбора метрик Postgres Pro и операционной системы.
oracle-fdw-std-10	Расширение oracle_fdw, предоставляющее обёртку сторонних данных для доступа к базам данных Oracle. Этот пакет предоставляется для систем RHEL, Ubuntu и SLES, Debian 10 и Astra Linux «Орёл» 2.12.
pg-repack-std-10	Расширение Postgres Pro и утилита для реорганизации таблиц.
pgpro-pgbadger	Анализатор журналов Postgres Pro, генерирующий подробные отчёты и графики.
pgbouncer	Пул соединений для Postgres Pro.
zstd	Утилиты командной строки для библиотеки libzstd.

Кроме того, Postgres Pro предоставляет отдельные пакеты с отладочной информацией для некоторых операционных систем:

- В системах на базе Debian это пакет postgrespro-std-10-dbg.
- В системах на базе RHEL это пакет postgrespro-std-10-debuginfo.
- В ОС Альт Линукс для всех пакетов с двоичными файлами имеются соответствующие пакеты -debuginfo.

Для серверных инсталляций требуются как минимум следующие пакеты:

- postgrespro-std-10-server
- postgrespro-std-10-client
- postgrespro-std-10-libs

Для использования дополнительных расширений Postgres Pro вы должны также установить пакет postgrespro-std-10-contrib. В системах на базе Debian пакет postgrespro-std-10-server зависит от пакета postgrespro-std-10-contrib, поэтому последний должен всегда устанавливаться вместе с сервером.

Для клиентских инсталляций обычно достаточно установить пакеты postgrespro-std-10-client и postgrespro-std-10-libs. Если вы используете независимые приложения и вам не нужны стандартные клиентские утилиты, такие как psql, вы можете установить только пакет postgrespro-std-10-libs.

Для рабочих станций разработчиков необходимый минимум составляют следующие пакеты:

- `postgrespro-std-10-libs`
- `postgrespro-std-10-devel`/ `postgrespro-std-10-dev`

Также вы можете установить и настроить сервер с тестовой базой данных. Подробнее о настройке, которая может потребоваться дополнительно, рассказывается в Подразделе 16.1.4.

16.1.3.2. Настройка автоматического запуска сервера

Если вы производите расширенную установку, автоматический запуск сервера по умолчанию отключается. После создания базы данных по умолчанию вы можете настроить автоматический запуск сервера при загрузке системы, используя имеющиеся в вашей операционной системе механизмы управления службами или сторонние средства обеспечения высокой степени доступности. Для облегчения этой задачи в пакете `postgrespro-std-10-server` поставляется скрипт pg-setup, помещаемый в каталог `/opt/pgpro/std-10/bin`.

Для настройки автозапуска сервера запустите скрипт pg-setup со следующими параметрами:

```
pg-setup service enable
```

Если требуется, вы можете отключить автозапуск сервера, используя тот же скрипт:

```
pg-setup service disable
```

Вы также можете использовать системные средства управления службами, непосредственно задействуя скрипты для систем инициализации SysV init.d и systemd, предоставляемые в пакете `postgrespro-std-10-server`. В зависимости от дистрибутива Linux Postgres Pro поддерживает следующие механизмы управления службами:

Дистрибутив Linux	**Предоставляемые скрипты**
RHEL 7/8 и совместимые дистрибутивы, SLES 12/15	Файл службы systemd
Debian, Ubuntu, Альт 8/9/10	И файл службы systemd, и скрипт init.d в стиле SysV

Для включения автоматического запуска сервера средствами systemd выполните следующую команду:

```
systemctl enable postgrespro-std-10
```

Для использования скрипта init.d в стиле SysV:

- В системах Альт Линукс выполните следующую команду:

  ```
  systemctl enable postgrespro-std-10
  ```

- В системах Debian воспользуйтесь скриптом `update-rc.d`. За подробностями обратитесь к соответствующей странице man.

16.1.4. Установка на рабочих станциях разработчиков

Хотя может быть достаточно установить пакеты `postgrespro-std-10-libs` и `postgrespro-std-10-devel`/`postgrespro-std-10-dev`, обычно на машинах разработчиков удобно иметь и установленный сервер. Для быстрой установки вы можете воспользоваться пакетом `postgrespro-std-10`, который автоматически настроит предоставляемые клиентские и серверные программы, а также создаст начальную базу данных. Однако если вы планируете использовать одновременно несколько продуктов на базе PostgreSQL, следуйте инструкциям по расширенной установке, приведённым в Подразделе 16.1.3.

Чтобы скомпилировать программы с библиотеками Postgres Pro, используя утилиту pg_config, поставляемую с Postgres Pro, необходимо, чтобы её путь был указан в переменной `PATH` до

путей каких-либо других версий pg_config. Заметьте, что в системах на базе RHEL pg_config не добавляется в `PATH` автоматически. Если в вашей системе нет других версий pg_config, вы можете воспользоваться утилитой pg-wrapper, поставляемой в пакете `postgrespro-std-10-client`, для создания символической ссылки на pg_config в стандартном каталоге исполняемых файлов.

Для компиляции программ с использованием pkg-config добавьте путь `/opt/pgpro/std-10/lib/pkgconfig/` в переменную среды `PKG_CONFIG_PATH`.

16.1.4.1. Использование сторонних программ с Postgres Pro

Чтобы использовать сервер Postgres Pro с клиентской программой из стороннего продукта, вы можете установить версию библиотек PostgreSQL, с которыми компилировалась эта программа. Например, если программа поставляется в составе ванильного PostgreSQL, вам может потребоваться установить пакеты `libpq` или `postgresql-libs`, предоставляемые для вашего дистрибутива Linux. В этом случае программа, возможно, не будет использовать некоторые новые возможности сервера Postgres Pro, но скорее всего она и не поддерживает их.

Если вы предпочитаете использовать библиотеки Postgres Pro со сторонней программой или хотели бы задействовать новую возможность, которая не требует изменения самого клиентского приложения, например, проверку подлинности SCRAM, вы можете перекомпилировать вашу программу с библиотеками Postgres Pro.

> **Важно**
>
> Если программа скомпилирована с одной версией libpq, а используется с другой, её стабильная работа не гарантируется.

Если вы создаёте пакеты `.rpm` или `.deb` для вашей программы, рекомендуется сделать следующее:

1. Добавьте путь `/opt/pgpro/std-10/bin` в переменную `PATH` в сборочных скриптах (в файле `.spec` или `debian/rules`).
2. Добавьте `postgrespro-std-10-dev` в теги `BuildDepends` или `BuildRequires` описания пакета вашей программы.

Тем самым вы обеспечите вызов в процессе создания пакетов нужной версии pg_config при каждой пересборке пакета исходного кода.

16.1.5. Настройка нескольких экземпляров Postgres Pro

Чтобы настроить в Linux несколько экземпляров сервера Postgres Pro с разными каталогами данных, нужно проделать следующее:

1. Установите и настройте Postgres Pro, следуя указаниям в Подразделе 16.1.2 или Подразделе 16.1.3.
2. После создания начальной базы данных запустите initdb и укажите путь к другому каталогу данных и любые другие параметры, требующиеся для инициализации другого экземпляра сервера.
3. Задайте другие порты для экземпляров серверов в соответствующих файлах `postgresql.conf`, чтобы не допустить конфликтов.
4. Если требуется, настройте автоматический запуск сервера следующим образом:
   a. Создайте копию файла `/etc/init.d/postgrespro-std-10` или `/lib/systemd/system/postgrespro-std-10.service` с другим именем и поменяйте в ней путь к каталогу данных.
   b. Включите автоматический запуск сервера, используя скрипты автозапуска, предоставленные для вашего системного механизма управления службами, вместо `pg-setup`, как описано в Подразделе 16.1.3.2. Убедитесь в том, что вы используете переименованные копии скриптов, созданные на шаге 1.

16.1.6. Замечания о совместимости с антивирусами

Настоятельно не рекомендуется использовать антивирусное ПО в системах, где работает Postgres Pro, так как антивирус создаёт дополнительную нагрузку и может оказывать непредсказуемое влияние на работу базы данных, что чревато проблемами с производительностью и надёжностью. Если вам необходимо использовать антивирусное ПО, обязательно добавьте в исключения антивируса следующие каталоги, так как они не содержат исполняемых файлов:

- Каталог `PGDATA` с основными данными кластера (обычно `/var/lib/pgpro/std-10/data` или другой каталог, заданный в параметрах initdb)
- Каталоги созданных табличных пространств

16.2. Установка Postgres Pro Standard в Windows

Postgres Pro предлагает следующие режимы установки в поддерживаемых системах Windows:

- Установка в графическом режиме с помощью интерактивного мастера
- Установка в командной строке

Вы также можете настроить среду для использования процедурных языков PL/Perl и PL/Python, если они требуются для ваших целей. За подробностями обратитесь к Подразделу 16.2.4.

Во избежание недопонимания учтите, что в системах Windows имеются следующие отдельные пользователи:

- Установка Postgres Pro производится от имени текущего пользователя операционной системы. Все файлы баз данных будут принадлежать этому пользователю.
- Чтобы работать с этой базой данных, вы должны будете подключиться к кластеру баз данных под именем пользователя, созданного во время установки. (Это не пользователь операционной системы.)
- По умолчанию служба Postgres Pro запускается от имени `NT AUTHORITY\NetworkService`, специального служебного пользователя Windows. При необходимости вы можете выбрать другого пользователя Windows, который будет запускать Postgres Pro, в соответствующем поле инсталлятора. Выбранный вами пользователь должен иметь право запуска служб Windows.

Минимальные системные требования для установки Postgres Pro, создания кластера баз данных и запуска сервера:

- Процессор с частотой 1 ГГц
- 1 ГБ ОЗУ
- 1 ГБ свободного дискового пространства

Для вспомогательных компонентов или данных потребуется дополнительное дисковое пространство.

16.2.1. Поддерживаемые версии Windows

Postgres Pro выпускается для следующих 64-битных версий Windows:

- Windows 10 и новее
- Windows Server 2012 R2 и новее

16.2.2. Установка с графическим интерфейсом

Postgres Pro предоставляет инсталлятор с интерактивным мастером, который устанавливает и конфигурирует ключевые компоненты Postgres Pro, создаёт начальную базу данных и настраивает автозапуск сервера. Для следующих компонентов предоставляются индивидуальные инсталляторы, которые должны запускаться отдельно после основного инсталлятора:

- `pg_probackup`

Для установки ключевых компонентов Postgres Pro запустите основной инсталлятор от имени администратора и следуйте инструкциям на экране. Учтите следующую специфику установки:

- Вы можете установить все основные компоненты Postgres Pro или настроить инсталляцию, исключив серверную часть или компоненты для разработчика. Клиентская часть устанавливается всегда.
- Выбранный каталог данных должен быть пустым. В противном случае Postgres Pro не сможет создать начальную базу данных. По умолчанию каталогом данных будет `C:\Program Files\PostgresPro\10\data`.
- Если установлен флажок Allow external connections (Разрешить внешние подключения) (отмеченный по умолчанию), инсталлятор вносит изменения в `postgresql.conf` и `pg_hba.conf` для разрешения подключений извне. В противном случае сервер Postgres Pro будет принимать подключения только с локального узла. Если вы решите разрешить внешние подключения после завершения установки, вы должны будете изменить конфигурацию Брандмауэра Windows, чтобы сервер Postgres Pro мог принимать подключения.
- Запомните имя и пароль пользователя, заданные вами при установке Postgres Pro, так как они потребуются для подключения к серверу Postgres Pro при использовании аутентификации с проверкой пароля.
- По умолчанию в Postgres Pro используются правила сортировки `icu`. Если вы обновляете инсталляцию, в которой использовались правила сортировки `libc`, например, инсталляцию PostgreSQL, выберите `libc` в качестве провайдера правил сортировки, воспользовавшись соответствующим выпадающим списком в инсталляторе.
- Postgres Pro предоставляет заранее настроенный ярлык psql, который можно найти в меню Пуск, чтобы вам не пришлось настраивать стандартные переменные окружения. Однако если вы хотели бы работать с Postgres Pro в обычной консоли командной строки, не задавая полные пути к двоичным файлам, отметьте флажок Set up environment variables (Настроить переменные окружения).

По завершении инсталляции экземпляр Postgres Pro готов к использованию, а для службы сервера включён автозапуск. Если вам понадобится больше одного экземпляра Postgres Pro в одной системе, вам нужно будет настроить дополнительные экземпляры вручную. За подробностями обратитесь к Подразделу 16.2.5.

16.2.3. Установка в командной строке

Чтобы установить Postgres Pro из командной строки, запустите загруженный файл инсталлятора, передав один или несколько параметров, описанных в Подразделе 16.2.3.1.

16.2.3.1. Параметры командной строки

Путь каталога инсталляции:

`/D=`*`путь`*

Полностью автоматическая установка:

`/S`

`INI`-файл, содержащий параметры установки сервера:

`/init=`*`имя_ini-файла`*

Если вы хотите скорректировать стандартные параметры, вы должны создать `INI`-файл вручную, как описано в Подразделе 16.2.3.2.

16.2.3.2. Формат INI-файла

В разделе [options] данного файла можно задать следующие параметры:

- `InstallDir` — путь для установки сервера. Если вы зададите параметр /D в командной строке, значение `InstallDir` будет иметь приоритет.
- `DataDir` — путь для создания баз данных по умолчанию

- `Port` — порт TCP/IP для приёма подключений. По умолчанию: 5432.
- `SuperUser` — имя пользователя, который получит права администратора баз данных
- `Password` — пароль этого пользователя
- `noExtConnections = 1` — не разрешать внешние подключения.
- `Coding = UNICODE` — кодировка символов для использования в базе данных
- `Locale` — локаль для использования в базе данных (для каждой кодировки может быть несколько локалей)
- `vcredist = no` — не устанавливать распространяемые библиотеки Visual C (это можно сделать, только если эти библиотеки уже установлены в системе)
- `envvar = 1` — установить полезные для Postgres Pro переменные окружения: `PGDATA`, `PGDATABASE`, `PGUSER`, `PGPORT`, `PGLOCALEDIR`
- `needoptimization = 0` — отключить автоматическую настройку параметров конфигурации в зависимости от имеющихся системных ресурсов.
- `datachecksums = 0` — отключить контрольные суммы в кластере.
- `serviceaccount` — пользователь Windows, от имени которого будет запускаться служба Postgres Pro. Указанный пользователь должен иметь право запуска служб Windows. По умолчанию служба Postgres Pro запускается от имени `NT AUTHORITY\NetworkService`, специальной учётной записи для служб Windows.
- `servicepassword` — пароль для пользователя Windows, указанного в параметре `serviceaccount`.
- `serviceid` — имя службы Postgres Pro.
- `islibc = 1` — использовать `libc` в качестве провайдера основного правила сортировки.

16.2.4. Загрузка процедурных языков

В дистрибутив Postgres Pro для систем Windows включены процедурные языки `PL/Perl` и `PL/Python`.

16.2.4.1. Настройка среды для PL/Python

Чтобы настроить систему для использования PL/Python, выполните следующие действия:

1. Загрузите и установите последнюю версию Python 3.8 или 2.7 для Windows. Выберите 64-битную версию, так как сервер Postgres Pro имеет 64-битную архитектуру. Вы можете найти подходящий инсталлятор на сайте *https://www.python.org/*.
2. Создайте расширение PL/Python, выполнив следующую команду в psql:

```
CREATE EXTENSION plpythonu
```

> **Подсказка**
>
> В некоторых случаях при создании функции на PL/Python соединение с сервером разрывается, а в журнал сервера выводится следующее сообщение об ошибке:
>
> `ImportError: module site not found`
>
> Для устранения этой проблемы попробуйте указать каталог инсталляции Python в переменной окружения `PYTHONHOME` и перезапустите службу Postgres Pro.

16.2.4.2. Настройка окружения для PL/Perl

Чтобы настроить систему для использования PL/Perl, выполните следующие действия:

1. Загрузите Strawberry Perl 5.26, выбрав на странице *https://strawberryperl.com/releases.html* MSI-пакет 64-битной версии, и установите его.
2. Перезапустите сервер Postgres Pro.
3. Создайте расширение PL/Perl, выполнив следующую команду в psql:

```
CREATE EXTENSION plperlu
```

16.2.5. Настройка нескольких экземпляров Postgres Pro

Чтобы настроить в Windows несколько экземпляров сервера Postgres Pro с разными каталогами данных, нужно проделать следующее:

1. Установите Postgres Pro, как описывается в Подразделе 16.2.2 или Подразделе 16.2.3. Установленные двоичные файлы используются всеми экземплярами Postgres Pro, поэтому данное действие нужно выполнить только один раз.
2. Выберите пустую папку, которую ваш новый экземпляр Postgres Pro будет использовать в качестве каталога данных. Например, это может быть `C:\Program Files\PostgresPro\10\data2`. К этой папке необходимо дать полный доступ текущему пользователю ОС, который будет владельцем файлов базы, и пользователю, от имени которого работает сервер (по умолчанию `NT AUTHORITY\NetworkService`).
3. Запустите программу initdb, передав ей полный путь к новому каталогу данных и другие параметры, требующиеся для инициализации нового экземпляра сервера. Например:

   ```
   "C:\Program Files\PostgresPro\10\bin\initdb.exe" --encoding=UTF8 -U "postgres" -D
    "C:\Program Files\PostgresPro\10\data2"
   ```

   Также вы можете остановить работающий сервер и скопировать содержимое существующего каталога данных в новый каталог. В этом случае новый экземпляр Postgres Pro наследует все свойства исходного экземпляра, включая параметры аутентификации.
4. Установите в `postgresql.conf` нового экземпляра Postgres Pro требуемые параметры. Обязательно назначьте отдельным экземплярам сервера разные порты во избежание конфликтов.
5. Откройте приглашение командной строки от имени администратора и зарегистрируйте новую службу Postgres Pro с уникальным именем, например, `postgrespro-data2`:

   ```
   "C:\Program Files\PostgresPro\10\bin\pg_ctl.exe" register -N "postgrespro-data2" -U
    "NT AUTHORITY\NetworkService" -D "C:\Program Files\PostgresPro\10\data2" -w
   ```

   Запустите зарегистрированную службу:

   ```
   sc start "postgrespro-data2"
   ```

После запуска службы ваш экземпляр Postgres Pro готов к использованию. Если вам нужны дополнительные расширения Postgres Pro, установите их в новом экземпляре, как описывается в Разделе 16.3.

16.2.6. Удаление Postgres Pro Standard

Процедура удаления Postgres Pro Standard зависит от количества экземпляров Postgres Pro, установленных в вашей системе.

Если у вас только один экземпляр Postgres Pro, созданный во время установки, вам будет достаточно запустить программу `C:\Program Files\PostgresPro\10\uninstall.exe`. Эта программа автоматически остановит сервер и разрегистрирует службу, связанную с этим экземпляром.

Если вы создавали дополнительные экземпляры Postgres Pro, вам нужно будет проделать следующие действия:

1. Остановите сервер для каждого экземпляра:

   ```
   "C:\Program Files\PostgresPro\10\bin\pg_ctl.exe" stop -D "C:\Program Files
   \PostgresPro\10\data_dir" -m fast -w
   ```

2. Разрегистрируйте службы для всех экземпляров:

   ```
   "C:\Program Files\PostgresPro\10\bin\pg_ctl.exe" unregister -N
    "имя_службы_postgrespro"
   ```

3. Запустите `C:\Program Files\PostgresPro\10\uninstall.exe`, чтобы удалить исполняемые файлы Postgres Pro.

16.2.7. Замечания о совместимости с антивирусами

Настоятельно не рекомендуется использовать антивирусное ПО в системах, где работает Postgres Pro, так как антивирус создаёт дополнительную нагрузку и может оказывать непредсказуемое влияние на работу базы данных, что чревато проблемами с производительностью и надёжностью. Если вам необходимо использовать антивирусное ПО, обязательно добавьте в исключения антивируса следующие каталоги, так как они не содержат исполняемых файлов:

- Каталог `PGDATA` с основными данными кластера (обычно `C:\Program Files\PostgresPro\10\data` или другой каталог, заданный в параметрах initdb)
- Каталоги созданных табличных пространств

16.3. Установка дополнительно поставляемых модулей

Postgres Pro поставляется с набором дополнительных серверных расширений, или модулей. В Linux они распространяются в отдельном пакете `postgrespro-contrib`. В Windows они устанавливаются вместе с серверными компонентами.

Установив двоичные файлы, вы должны развернуть в базе данных дополнительные нужные вам расширения. В большинстве случае для этого достаточно выполнить команду CREATE EXTENSION. Однако для некоторых расширений также требуется, чтобы при запуске сервера загружались определённые разделяемые библиотеки. Если вы хотите использовать такие расширения, вам нужно настроить параметр

```
shared_preload_libraries = 'lib1, lib2, lib3'
```

в файле `postgresql.conf` вашего экземпляра Postgres Pro и перезапустить сервер, а затем выполнить команду `CREATE EXTENSION`.

Более конкретные инструкции по установке и настройке определённого расширения вы можете найти в его документации в разделе Приложение F.

Чтобы получить список расширений, доступных в вашей инсталляции Postgres Pro, просмотрите системный каталог `pg_available_extensions`.

16.4. Миграция на Postgres Pro

Разные основные версии Postgres Pro, а также разные продукты на базе PostgreSQL, имеющие одинаковую основную версию, могут иметь несовместимые на двоичном уровне базы данных, так что просто заменить программу сервера и продолжить использовать его в общем случае нельзя. Чтобы преобразовать базы данных с более старой основной версии, вам потребуется выполнить выгрузку/загрузку данных с помощью pg_dumpall или использовать pg_upgrade.

Инструкции по обновлению для определённой версии вы найдёте в Замечаниях к выпуску для этой версии Postgres Pro.

Глава 17. Подготовка к работе и сопровождение сервера

В этой главе рассказывается, как установить и запустить сервер баз данных, и о том, как он взаимодействует с операционной системой.

17.1. Учётная запись пользователя Postgres Pro

Как и любую другую службу, доступную для внешнего мира, Postgres Pro рекомендуется запускать под именем отдельного пользователя. Эта учётная запись должна владеть только данными, которыми управляет сервер, и разделять её с другими службами не следует. (Например, не стоит использовать для этого пользователя `nobody`.) Также рекомендуется не устанавливать под именем этого пользователя исполняемые файлы, чтобы их нельзя было подменить в случае компрометации системы.

Для создания пользователя в Unix-подобной системе следует искать команду `useradd` или `adduser`. В качестве имени пользователя часто используется postgres, и именно это имя предполагается в данной документации, но вы можете выбрать и другое, если захотите.

17.2. Создание кластера баз данных

Прежде чем вы сможете работать с базами данных, вы должны проинициализировать область хранения баз данных на диске. Мы называем это хранилище *кластером баз данных*. (В SQL применяется термин «кластер каталога».) Кластер баз данных представляет собой набор баз, управляемых одним экземпляром работающего сервера. После инициализации кластер будет содержать базу данных с именем `postgres`, предназначенную для использования по умолчанию утилитами, пользователями и сторонними приложениями. Сам сервер баз данных не требует наличия базы `postgres`, но многие внешние вспомогательные программы рассчитывают на её существование. При инициализации в каждом кластере создаётся ещё одна база, с именем `template1`. Как можно понять из имени, она применяется впоследствии в качестве шаблона создаваемых баз данных; использовать её в качестве рабочей не следует. (За информацией о создании новых баз данных в кластере обратитесь к Главе 21.)

С точки зрения файловой системы, кластер баз данных представляет собой один каталог, в котором будут храниться все данные. Мы называем его *каталогом данных* или *областью данных*. Где именно хранить данные, вы абсолютно свободно можете выбирать сами. Какого-либо стандартного пути не существует, но часто данные размещаются в `/usr/local/pgsql/data` или в `/var/lib/pgsql/data`. Для инициализации кластера баз данных применяется команда initdb, которая устанавливается в составе Postgres Pro. Расположение кластера базы данных в файловой системе задаётся параметром `-D`, например:

```
$ initdb -D /usr/local/pgsql/data
```

Заметьте, что эту команду нужно выполнять от имени учётной записи Postgres Pro, о которой говорится в предыдущем разделе.

> **Подсказка**
>
> В качестве альтернативы параметра `-D` можно установить переменную окружения `PGDATA`.

Также можно запустить команду `initdb`, воспользовавшись программой pg_ctl , примерно так:

```
$ pg_ctl -D /usr/local/pgsql/data initdb
```

Этот вариант может быть удобнее, если вы используете `pg_ctl` для запуска и остановки сервера (см. Раздел 17.3), так как `pg_ctl` будет единственной командой, с помощью которой вы будете управлять экземпляром сервера баз данных.

Команда initdb попытается создать указанный вами каталог, если он не существует. Конечно, она не сможет это сделать, если initdb не будет разрешено записывать в родительский каталог. Вообще рекомендуется, чтобы пользователь Postgres Pro был владельцем не только каталога данных, но и родительского каталога, так что такой проблемы быть не должно. Если же и нужный родительский каталог не существует, вам нужно будет сначала создать его, используя права root, если вышестоящий каталог защищён от записи. Таким образом, процедура может быть такой:

```
root# mkdir /usr/local/pgsql
root# chown postgres /usr/local/pgsql
root# su postgres
postgres$ initdb -D /usr/local/pgsql/data
```

Команда initdb не будет работать, если указанный каталог данных уже существует и содержит файлы; это мера предохранения от случайной перезаписи существующей инсталляции.

Так как каталог данных содержит все данные базы, очень важно защитить его от неавторизованного доступа. Для этого initdb лишает прав доступа к нему всех пользователей, кроме пользователя Postgres Pro.

Однако даже когда содержимое каталога защищено, если проверка подлинности клиентов настроена по умолчанию, любой локальный пользователь может подключиться к базе данных и даже стать суперпользователем. Если вы не доверяете другим локальным пользователям, мы рекомендуем использовать один из параметров команды initdb: -W, --pwprompt или --pwfile и назначить пароль суперпользователя баз данных. Кроме того, воспользуйтесь параметром -A md5 или -A password и отключите разрешённый по умолчанию режим аутентификации trust; либо измените сгенерированный файл pg_hba.conf после выполнения initdb, но *перед* тем, как запустить сервер в первый раз. (Возможны и другие разумные подходы — применить режим проверки подлинности peer или ограничить подключения на уровне файловой системы. За дополнительными сведениями обратитесь к Главе 19.)

Команда initdb также устанавливает для кластера баз данных локаль по умолчанию. Обычно она просто берёт параметры локали из текущего окружения и применяет их к инициализируемой базе данных. Однако можно выбрать и другую локаль для базы данных; за дополнительной информацией обратитесь к Разделу 22.1. Команда initdb задаёт порядок сортировки по умолчанию для применения в определённом кластере баз данных, и хотя новые базы данных могут создаваться с иным порядком сортировки, порядок в базах-шаблонах, создаваемых initdb, можно изменить, только если удалить и пересоздать их. Также учтите, что при использовании локалей, отличных от C и POSIX, возможно снижение производительности. Поэтому важно правильно выбрать локаль с самого начала.

Команда initdb также задаёт кодировку символов по умолчанию для кластера баз данных. Обычно она должна соответствовать кодировке локали. За подробностями обратитесь к Разделу 22.3.

Для локалей, отличных от C и POSIX, порядок сортировки символов зависит от системной библиотеки локализации, а он, в свою очередь, влияет на порядок ключей в индексах. Поэтому кластер нельзя перевести на несовместимую версию библиотеки ни путём восстановления снимка, ни через двоичную репликацию, ни перейдя на другую операционную систему или обновив её версию.

17.2.1. Использование дополнительных файловых систем

Во многих инсталляциях кластеры баз данных создаются не в «корневом» томе, а в отдельных файловых системах (томах). Если вы решите сделать так же, то не следует выбирать в качестве каталога данных самый верхний каталог дополнительного тома (точку монтирования). Лучше всего создать внутри каталога точки монтирования каталог, принадлежащий пользователю Postgres Pro, а затем создать внутри него каталог данных. Это исключит проблемы с разрешениями, особенно для таких операций, как pg_upgrade, и при этом гарантирует чистое поведение в случае, если дополнительный том окажется отключён.

17.2.2. Использование сетевых файловых систем

Во многих инсталляциях кластеры баз данных создаются в сетевых файловых ресурсах. Иногда это реализуется с применением сетевой файловой системы (NFS, Network File System) или сетевых хранилищ (NAS, Network Attached Storage), использующих NFS внутри. Postgres Pro не делает ничего специфического с файловыми системами NFS, то есть он предполагает, что NFS работает точно так же, как и локально подключённые диски. Но если реализация клиента или сервера NFS не обеспечивает стандартное поведение файловой системы, это чревато нестабильной работой (см. *http://www.time-travellers.org/shane/papers/NFS_considered_harmful.html*). В частности, возможно разрушение данных при отложенной (асинхронной) записи на сервер NFS. Поэтому, по возможности, во избежание таких проблем монтируйте файловые системы NFS в синхронном режиме (без кеширования). Кроме того, не рекомендуется применять мягкое монтирование файловой системы NFS.

В сетях хранения данных (SAN, Storage Area Networks) обычно используются собственные протоколы, не NFS, и они могут быть не подвержены (а могут быть и подвержены) этим рискам. По вопросам гарантии согласованности данных обратитесь к документации производителя. Postgres Pro не может быть надёжнее файловой системы, которую он использует.

17.3. Запуск сервера баз данных

> **Важно**
>
> При установке двоичных пакетов в системах Linux базы данных по умолчанию располагаются в каталоге `/var/lib/pgpro/std-10/data`, если явно не будет задан другой каталог. За подробностями обратитесь к Разделу 16.1.

Чтобы кто-либо смог обратиться к базе данных, необходимо сначала запустить сервер баз данных. Программа сервера называется `postgres`. Для работы программа `postgres` должна знать, где найти данные, которые она будет использовать. Указать это местоположение позволяет параметр `-D`. Таким образом, проще всего запустить сервер, выполнив команду:

```
$ postgres -D /usr/local/pgsql/data
```

в результате которой сервер продолжит работу в качестве процесса переднего плана. Запускать эту команду следует под именем учётной записи Postgres Pro. Без параметра `-D` сервер попытается использовать каталог данных, указанный в переменной окружения `PGDATA`. Если и эта переменная не определена, сервер не будет запущен.

Однако обычно лучше запускать `postgres` в фоновом режиме. Для этого можно применить обычный синтаксис, принятый в оболочке Unix:

```
$ postgres -D /usr/local/pgsql/data >logfile 2>&1 &
```

Важно где-либо сохранять информацию, которую выводит сервер в каналы stdout и stderr, как показано выше. Это полезно и для целей аудита, и для диагностики проблем. (Более глубоко работа с файлами журналов рассматривается в Разделе 23.3.)

Программа `postgres` также принимает несколько других параметров командной строки. За дополнительными сведениями обратитесь к справочной странице postgres и к следующей Главе 18.

Такой вариант запуска довольно быстро может оказаться неудобным. Поэтому для упрощения подобных задач предлагается вспомогательная программа pg_ctl. Например:

```
pg_ctl start -l logfile
```

запустит сервер в фоновом режиме и направит выводимые сообщения сервера в указанный файл журнала. Параметр `-D` для неё имеет то же значение, что и для программы `postgres`. С помощью `pg_ctl` также можно остановить сервер.

Обычно возникает желание, чтобы сервер баз данных сам запускался при загрузке операционной системы. Скрипты автозапуска для разных систем разные, но в составе Postgres Pro предлагается

несколько типовых скриптов в каталоге contrib/start-scripts. Для установки такого скрипта в систему требуются права root.

В различных системах приняты разные соглашения о порядке запуска служб в процессе загрузки. Во многих системах для этого используется файл /etc/rc.local или /etc/rc.d/rc.local. В других применяются каталоги init.d или rc.d. Однако при любом варианте запускаться сервер должен от имени пользователя Postgres Pro, но *не root* или какого-либо другого пользователя. Поэтому команду запуска обычно следует записывать в форме su postgres -c '...'. Например:

```
su postgres -c 'pg_ctl start -D /usr/local/pgsql/data -l serverlog'
```

Ниже приведены более конкретные предложения для нескольких основных ОС. (Вместо указанных нами шаблонных значений необходимо подставить правильный путь к каталогу данных и фактическое имя пользователя.)

- Для запуска во FreeBSD воспользуйтесь файлом contrib/start-scripts/freebsd в дереве исходного кода Postgres Pro.
- В OpenBSD, добавьте в файл /etc/rc.local следующие строки:

  ```
  if [ -x /usr/local/pgsql/bin/pg_ctl -a -x /usr/local/pgsql/bin/postgres ]; then
      su -l postgres -c '/usr/local/pgsql/bin/pg_ctl start -s -l /var/postgresql/log -
  D /usr/local/pgsql/data'
      echo -n ' postgresql'
  fi
  ```

- В системах Linux вы можете либо добавить

  ```
  /usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data
  ```

  в /etc/rc.d/rc.local или в /etc/rc.local, либо воспользоваться файлом contrib/start-scripts/linux в дереве исходного кода Postgres Pro.

  Используя systemd, вы можете применить следующий файл описания службы (например, /etc/systemd/system/postgresql.service):

  ```
  [Unit]
  Description=Postgres Pro database server
  Documentation=man:postgres(1)
  After=network-online.target
  Wants=network-online.target

  [Service]
  Type=notify
  User=postgres
  ExecStart=/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
  ExecReload=/bin/kill -HUP $MAINPID
  KillMode=mixed
  KillSignal=SIGINT
  TimeoutSec=infinity

  [Install]
  WantedBy=multi-user.target
  ```

  Для использования Type=notify требуется, чтобы сервер был скомпилирован с указанием configure --with-systemd.

  Особого внимания заслуживает значение тайм-аута. На момент написания этой документации по умолчанию в systemd принят тайм-аут 90 секунд, так что процесс, не сообщивший о своей готовности за это время, будет уничтожен. Но серверу Postgres Pro при запуске может потребоваться выполнить восстановление после сбоя, так что переход в состояние готовности может занять гораздо больше времени. Предлагаемое значение infinity отключает логику тайм-аута.

- В NetBSD можно использовать скрипт запуска для FreeBSD или для Linux, в зависимости от предпочтений.
- В Solaris, создайте файл с именем `/etc/init.d/postgresql`, содержащий следующую стоку:

  ```
  su - postgres -c "/usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data"
  ```

  Затем создайте символическую ссылку на него в каталоге `/etc/rc3.d` с именем `S99postgresql`.

Когда сервер работает, идентификатор его процесса (PID) сохраняется в файле `postmaster.pid` в каталоге данных. Это позволяет исключить запуск нескольких экземпляров сервера с одним каталогом данных, а также может быть полезно для выключения сервера.

17.3.1. Сбои при запуске сервера

Есть несколько распространённых причин, по которым сервер может не запуститься. Чтобы понять, чем вызван сбой, просмотрите файл журнала сервера или запустите сервер вручную (не перенаправляя его потоки стандартного вывода и ошибок) и проанализируйте выводимые сообщения. Ниже мы рассмотрим некоторые из наиболее частых сообщений об ошибках более подробно.

```
LOG:  could not bind IPv4 address "127.0.0.1": Address already in use
HINT:  Is another postmaster already running on port 5432? If not, wait a few seconds
 and retry.
FATAL:  could not create any TCP/IP sockets
```

Это обычно означает именно то, что написано: вы пытаетесь запустить сервер на том же порту, на котором уже работает другой. Однако если сообщение ядра не `Address already in use` или подобное, возможна и другая проблема. Например, при попытке запустить сервер с номером зарезервированного порта будут выданы такие сообщения:

```
$ postgres -p 666
LOG:  could not bind IPv4 address "127.0.0.1": Permission denied
HINT:  Is another postmaster already running on port 666? If not, wait a few seconds
 and retry.
FATAL:  could not create any TCP/IP sockets
```

Следующее сообщение:

```
FATAL:  could not create shared memory segment: Invalid argument
DETAIL:  Failed system call was shmget(key=5440001, size=4011376640, 03600).
```

может означать, что установленный для вашего ядра предельный размер разделяемой памяти слишком мал для рабочей области, которую пытается создать Postgres Pro (в данном примере 4011376640 байт). Возможно также, что в вашем ядре вообще отсутствует поддержка разделяемой памяти в стиле System-V. В качестве временного решения можно попытаться запустить сервер с меньшим числом буферов (shared_buffers), но в итоге вам, скорее всего, придётся переконфигурировать ядро и увеличить допустимый размер разделяемой памяти. Вы также можете увидеть это сообщение при попытке запустить несколько серверов на одном компьютере, если запрошенный ими объём разделяемой памяти в сумме превышает этот предел.

Сообщение:

```
FATAL:  could not create semaphores: No space left on device
DETAIL:  Failed system call was semget(5440126, 17, 03600).
```

не означает, что у вас закончилось место на диске. Это значит, что установленное в вашем ядре предельное число семафоров System V меньше, чем количество семафоров, которое пытается создать Postgres Pro. Как и в предыдущем случае можно попытаться обойти эту проблему, запустив сервер с меньшим числом допустимых подключений (max_connections), но в конце концов вам придётся увеличить этот предел в ядре.

Если вы получаете ошибку «illegal system call» (неверный системный вызов), то, вероятнее всего, ваше ядро вовсе не поддерживает разделяемую память или семафоры. В этом случае вам остаётся только переконфигурировать ядро и включить их поддержку.

Настройка средств IPC в стиле System V описывается в Подразделе 17.4.1.

17.3.2. Проблемы с подключениями клиентов

Хотя ошибки подключений, возможные на стороне клиента, довольно разнообразны и зависят от приложений, всё же несколько проблем могут быть связаны непосредственно с тем, как был запущен сервер. Описание ошибок, отличных от описанных ниже, следует искать в документации соответствующего клиентского приложения.

```
psql: could not connect to server: Connection refused
        Is the server running on host "server.joe.com" and accepting
        TCP/IP connections on port 5432?
```

Это общая проблема «я не могу найти сервер и начать взаимодействие с ним». Показанное выше сообщение говорит о попытке установить подключение по TCP/IP. Очень часто объясняется это тем, что сервер просто забыли настроить для работы по протоколу TCP/IP.

Кроме того, при попытке установить подключение к локальному серверу через Unix-сокет можно получить такое сообщение:

```
psql: could not connect to server: No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

Путь в последней строке помогает понять, к правильному ли адресу пытается подключиться клиент. Если сервер на самом деле не принимает подключения по этому адресу, обычно выдаётся сообщение ядра `Connection refused` (В соединении отказано) или `No such file or directory` (Нет такого файла или каталога), приведённое выше. (Важно понимать, что `Connection refused` в данном контексте *не* означает, что сервер получил запрос на подключение и отверг его. В этом случае были бы выданы другие сообщения, например, показанные в Разделе 19.4.) Другие сообщения об ошибках, например `Connection timed out` (Тайм-аут соединения) могут сигнализировать о более фундаментальных проблемах, например, о нарушениях сетевых соединений.

17.4. Управление ресурсами ядра

Postgres Pro иногда может исчерпывать некоторые ресурсы операционной системы до предела, особенно при запуске нескольких копий сервера в одной системе или при работе с очень большими базами. В этом разделе описываются ресурсы ядра, которые использует Postgres Pro, и подходы к решению проблем, связанных с ограниченностью этих ресурсов.

17.4.1. Разделяемая память и семафоры

Postgres Pro требует, чтобы операционная система предоставляла средства межпроцессного взаимодействия (IPC), в частности, разделяемую память и семафоры. Системы семейства Unix обычно предоставляют функции IPC в стиле «System V» или функции IPC в стиле «POSIX» или и те, и другие. В Windows эти механизмы реализованы по-другому, но здесь это не рассматривается.

Если эти механизмы полностью отсутствуют в системе, при запуске сервера обычно выдаётся ошибка «Illegal system call» (Неверный системный вызов). В этом случае единственный способ решить проблему — переконфигурировать ядро системы. Без них Postgres Pro просто не будет работать. Это довольно редкая ситуация, особенно с современными операционными системами.

При запуске сервера Postgres Pro обычно запрашивает очень небольшой объём разделяемой памяти System V и намного больший объём памяти POSIX (`mmap`). Помимо этого при запуске создаётся значительное количество семафоров (в стиле System V или POSIX). В настоящее время

семафоры POSIX используются в системах Linux и FreeBSD, а на других платформах используются семафоры System V.

> **Примечание**
>
> PostgreSQL до версии 9.3 использовал только разделяемую память System V, поэтому необходимый для запуска сервера объём разделяемой памяти System V был гораздо больше. Если вы используете более раннюю версию сервера, обратитесь к документации по вашей версии.

Функции IPC в стиле System V обычно сталкиваются с лимитами на уровне системы. Когда Postgres Pro превышает один из этих лимитов, сервер отказывается запускаться, но должен выдать полезное сообщение, говорящее об ошибке и о том, что с ней делать. (См. также Подраздел 17.3.1.) Соответствующие параметры ядра в разных системах называются аналогично (они перечислены в Таблице 17.1), но устанавливаются по-разному. Ниже предлагаются способы их изменения для некоторых систем.

Таблица 17.1. Параметры IPC в стиле System V

Имя	**Описание**	**Значения, необходимые для запуска одного экземпляра Postgres Pro**
SHMMAX	Максимальный размер сегмента разделяемой памяти (в байтах)	как минимум 1 КБ, но значение по умолчанию обычно гораздо больше
SHMMIN	Минимальный размер сегмента разделяемой памяти (в байтах)	1
SHMALL	Общий объём доступной разделяемой памяти (в байтах или страницах)	если в байтах, то же, что и SHMMAX; если в страницах, то ceil(SHMMAX/PAGE_SIZE) , плюс потребность других приложений
SHMSEG	Максимальное число сегментов разделяемой памяти для процесса	требуется только 1 сегмент, но значение по умолчанию гораздо больше
SHMMNI	Максимальное число сегментов разделяемой памяти для всей системы	как SHMSEG плюс потребность других приложений
SEMMNI	Максимальное число идентификаторов семафоров (т. е., их наборов)	как минимум ceil((max_connections + autovacuum_max_workers + max_worker_processes + 5) / 16) плюс потребность других приложений
SEMMNS	Максимальное число семафоров для всей системы	ceil((max_connections + autovacuum_max_workers + max_worker_processes + 5) / 16) * 17 плюс потребность других приложений
SEMMSL	Максимальное число семафоров в наборе	не меньше 17
SEMMAP	Число записей в карте семафоров	см. текст

Имя	Описание	Значения, необходимые для запуска одного экземпляра Postgres Pro
SEMVMX	Максимальное значение семафора	не меньше 1000 (по умолчанию оно обычно равно 32767; без необходимости менять его не следует)

Postgres Pro запрашивает небольшой блок разделяемой памяти System V (обычно 48 байт на 64-битной платформе) для каждой копии сервера. В большинстве современных операционных систем такой объём выделяется без проблем. Однако если запускать много копий сервера или разделяемую память System V занимают и другие приложения, может понадобиться увеличить значение SHMALL, задающее общий объём разделяемой памяти System V, доступный для всей системы. Заметьте, что SHMALL во многих системах задаётся в страницах, а не в байтах.

Менее вероятны проблемы с минимальным размером сегментов разделяемой памяти (SHMMIN), который для Postgres Pro не должен превышать примерно 32 байт (обычно это всего 1 байт). Максимальное число сегментов для всей системы (SHMMNI) или для одного процесса (SHMSEG) тоже обычно не влияет на работоспособность сервера, если только это число не равно нулю.

Когда Postgres Pro использует семафоры System V, он занимает по одному семафору на одно разрешённое подключение (max_connections), на разрешённый рабочий процесс автоочистки (autovacuum_max_workers) и фоновый процесс (max_worker_processes), в наборах по 16. В каждом таком наборе есть также 17-ый семафор, содержащий «магическое число», позволяющий обнаруживать коллизии с наборами семафоров других приложений. Максимальное число семафоров в системе задаётся параметром SEMMNS, который, следовательно, должен быть равен как минимум сумме max_connections, autovacuum_max_workers и max_worker_processes, плюс один дополнительный на каждые 16 семафоров подключений и рабочих процессов (см. формулу в Таблице 17.1). Параметр SEMMNI определяет максимальное число наборов семафоров, которые могут существовать в системе в один момент времени. Таким образом, его значение должно быть не меньше чем ceil((max_connections + autovacuum_max_workers + max_worker_processes + 5) / 16). В качестве временного решения проблем, которые вызываются этими ограничениями, но обычно сопровождаются некорректными сообщениями функции semget, например, «No space left on device» (На устройстве не осталось места), можно уменьшить число разрешённых соединений.

В некоторых случаях может потребоваться увеличить SEMMAP как минимум до уровня SEMMNS. Если в системе есть такой параметр (а во многих системах его нет), он определяет размер карты ресурсов семафоров, в которой выделяется запись для каждого непрерывного блока семафоров. Когда набор семафоров освобождается, эта запись либо добавляется к существующей соседней записи, либо регистрируется как новая запись в карте. Если карта переполняется, освобождаемые семафоры теряются (до перезагрузки). Таким образом, фрагментация пространства семафоров может со временем привести к уменьшению числа доступных семафоров.

Другие параметры, связанные с «аннулированием операций» с семафорами, например, SEMMNU и SEMUME, на работу Postgres Pro не влияют.

При использовании семафоров POSIX требуемое их количество не отличается от количества для System V, то есть по одному семафору на разрешённое подключение (max_connections), на разрешённый рабочий процесс автоочистки (autovacuum_max_workers) и фоновый процесс (max_worker_processes). На платформах, где предпочитается этот вариант, отсутствует определённый лимит ядра на количество семафоров POSIX.

AIX

> Как минимум с версии 5.1, для таких параметров, как SHMMAX, никакая дополнительная настройка не должна требоваться, так как система, похоже, позволяет использовать всю память в качестве разделяемой. Подобная конфигурация требуется обычно и для других баз данных, например, для DB/2.

Однако может понадобиться изменить глобальные параметры `ulimit` в `/etc/security/limits`, так как стандартные жёсткие ограничения на размер (`fsize`) и количество файлов (`nofiles`) могут быть недостаточно большими.

FreeBSD

Значения параметров IPC по умолчанию можно изменить, используя возможности `sysctl` или `loader`. С помощью `sysctl` можно задать следующие параметры:

```
# sysctl kern.ipc.shmall=32768
# sysctl kern.ipc.shmmax=134217728
```

Чтобы эти изменения сохранялись после перезагрузки, измените `/etc/sysctl.conf`.

Эти параметры, связанные с семафорами, `sysctl` менять не позволяет, но их можно задать в `/boot/loader.conf`:

```
kern.ipc.semmni=256
kern.ipc.semmns=512
```

Чтобы изменённые таким образом параметры вступили в силу, требуется перезагрузить систему.

Возможно, вы захотите настроить ядро так, чтобы разделяемая память всегда находилась в ОЗУ и никогда не выгружалась в пространство подкачки. Это можно сделать, установив с помощью `sysctl` параметр `kern.ipc.shm_use_phys`.

Если вы используете «камеры» FreeBSD, включив в sysctl параметр `security.jail.sysvipc_allowed`, главные процессы postmaster, работающие в разных камерах, должны запускаться разными пользователями операционной системы. Это усиливает защиту, так как не позволяет обычным пользователям обращаться к разделяемой памяти или семафорам в разных камерах, и при этом способствует корректной работе кода очистки IPC в Postgres Pro. (Во FreeBSD 6.0 и более поздних версиях код очистки IPC не может корректно выявить процессы в других камерах, что не позволяет запускать процессы postmaster на одном порту в разных камерах.)

До версии 4.0 система FreeBSD работала так же, как и старая OpenBSD (см. ниже).

NetBSD

В NetBSD, начиная с версии 5.0, параметры IPC можно изменить, воспользовавшись командой `sysctl`, например:

```
$ sysctl -w kern.ipc.semmni=100
```

Чтобы эти параметры сохранялись после перезагрузки, измените `/etc/sysctl.conf`.

Обычно имеет смысл увеличить `kern.ipc.semmni` и `kern.ipc.semmns`, так как их значения по умолчанию в NetBSD слишком малы.

Возможно, вы захотите настроить ядро так, чтобы разделяемая память всегда находилась в ОЗУ и никогда не выгружалась в пространство подкачки. Это можно сделать, установив с помощью `sysctl` параметр `kern.ipc.shm_use_phys`.

До версии 5.0 система NetBSD работала так же, как старые OpenBSD (см. ниже), за исключением того, что параметры ядра в этой системе устанавливаются с указанием `options`, а не `option`.

OpenBSD

В OpenBSD, начиная с версии 3.3, параметры IPC можно изменить, воспользовавшись командой `sysctl`, например:

```
$ sysctl kern.seminfo.semmni=100
```

Чтобы эти параметры сохранялись после перезагрузки, измените `/etc/sysctl.conf`.

Обычно имеет смысл увеличить `kern.seminfo.semmni` и `kern.seminfo.semmns`, так как их значения по умолчанию в OpenBSD слишком малы.

В старых версиях OpenBSD вам потребуется пересобрать ядро, чтобы изменить параметры IPC. Также убедитесь, что в ядре включены параметры `SYSVSHM` и `SYSVSEM` (по умолчанию они включены). Следующие строки показывают, как установить различные параметры в файле конфигурации ядра:

```
option        SYSVSHM
option        SHMMAXPGS=4096
option        SHMSEG=256

option        SYSVSEM
option        SEMMNI=256
option        SEMMNS=512
option        SEMMNU=256
```

HP-UX

Значения по умолчанию обычно вполне удовлетворяют средним потребностям. В HP-UX 10 параметр `SEMMNS` по умолчанию имеет значение 128, что может быть недостаточно для больших баз данных.

Параметры IPC можно установить в менеджере системного администрирования (System Administration Manager, SAM) в разделе Kernel Configuration (Настройка ядра) → Configurable Parameters (Настраиваемые параметры). Установив нужные параметры, выполните операцию Create A New Kernel (Создать ядро).

Linux

По умолчанию максимальный размер сегмента равен 32 МБ, а максимальный общий размер составляет 2097152 страниц. Страница почти всегда содержит 4096 байт, за исключением нестандартных конфигураций ядра с поддержкой «огромных страниц» (точно узнать размер страницы можно, выполнив `getconf PAGE_SIZE`).

Параметры размера разделяемой памяти можно изменить, воспользовавшись командой `sysctl`. Например, так можно выделить 16 ГБ для разделяемой памяти:

```
$ sysctl -w kernel.shmmax=17179869184
$ sysctl -w kernel.shmall=4194304
```

Чтобы сохранить эти изменения после перезагрузки, их также можно записать в файл `/etc/sysctl.conf` (это настоятельно рекомендуется).

В некоторых старых дистрибутивах может не оказаться программы `sysctl`, но те же изменения можно произвести, обратившись к файловой системе `/proc`:

```
$ echo 17179869184 >/proc/sys/kernel/shmmax
$ echo 4194304 >/proc/sys/kernel/shmall
```

Остальные параметры имеют вполне подходящие значения, так что их обычно менять не нужно.

macOS

Для настройки разделяемой памяти в macOS рекомендуется создать файл `/etc/sysctl.conf` и записать в него присваивания переменных следующим образом:

```
kern.sysv.shmmax=4194304
kern.sysv.shmmin=1
```

```
kern.sysv.shmmni=32
kern.sysv.shmseg=8
kern.sysv.shmall=1024
```

Заметьте, что в некоторых версиях macOS, *все пять* параметров разделяемой памяти должны быть установлены в `/etc/sysctl.conf`, иначе их значения будут проигнорированы.

Имейте в виду, что последние версии macOS игнорируют попытки задать для `SHMMAX` значение, не кратное 4096.

`SHMALL` на этой платформе измеряется в страницах (по 4 КБ).

В старых версиях macOS, чтобы изменения параметров разделяемой памяти вступили в силу, требовалась перезагрузка. Начиная с версии 10.5, все параметры, кроме `SHMMNI` можно изменить «на лету», воспользовавшись командой sysctl. Но тем не менее лучше задавать выбранные вами значения в `/etc/sysctl.conf`, чтобы они сохранялись после перезагрузки.

Файл `/etc/sysctl.conf` обрабатывается только начиная с macOS версии 10.3.9. Если вы используете предыдущий выпуск 10.3.x, необходимо отредактировать файл `/etc/rc` и задать значения следующими командами:

```
sysctl -w kern.sysv.shmmax
sysctl -w kern.sysv.shmmin
sysctl -w kern.sysv.shmmni
sysctl -w kern.sysv.shmseg
sysctl -w kern.sysv.shmall
```

Заметьте, что `/etc/rc` обычно заменяется при обновлении системы macOS, так что следует ожидать, что вам придётся повторять эти изменения после каждого обновления.

В macOS 10.2 и более ранних версиях вместо этого надо записать эти команды в файле `/System/Library/StartupItems/SystemTuning/SystemTuning`.

Solaris версии с 2.6 по 2.9 (Solaris 6 .. Solaris 9)

Соответствующие параметры можно изменить в `/etc/system`, например так:

```
set shmsys:shminfo_shmmax=0x2000000
set shmsys:shminfo_shmmin=1
set shmsys:shminfo_shmmni=256
set shmsys:shminfo_shmseg=256

set semsys:seminfo_semmap=256
set semsys:seminfo_semmni=512
set semsys:seminfo_semmns=512
set semsys:seminfo_semmsl=32
```

Чтобы изменения вступили в силу, потребуется перегрузить систему. Информацию о разделяемой памяти в более старых версиях Solaris можно найти по ссылке *http://sunsite.uakom.sk/sunworldonline/swol-09-1997/swol-09-insidesolaris.html*.

Solaris 2.10 (Solaris 10) и более поздние версии
OpenSolaris

В Solaris 10 и новее, а также в OpenSolaris, стандартные параметры разделяемой памяти и семафоров достаточно хороши для большинства применений Postgres Pro. По умолчанию Solaris теперь устанавливает в `SHMMAX` четверть объёма ОЗУ. Чтобы изменить этот параметр, воспользуйтесь возможностью задать параметр проекта, связанного с пользователем `postgres`. Например, выполните от имени `root` такую команду:

```
projadd -c "Postgres Pro DB User" -K "project.max-shm-memory=(privileged,8GB,deny)"
 -U postgres -G postgres user.postgres
```

Эта команда создаёт проект `user.postgres` и устанавливает максимальный объём разделяемой памяти для пользователя `postgres` равным 8 ГБ. Это изменение вступает в силу при следующем входе этого пользователя или при перезапуске Postgres Pro (не перезагрузке конфигурации). При этом подразумевается, что Postgres Pro выполняется пользователем `postgres` в группе `postgres`. Перезагружать систему после этой команды не нужно.

Для серверов баз данных, рассчитанных на большое количество подключений, рекомендуется также изменить следующие параметры:

```
project.max-shm-ids=(priv,32768,deny)
project.max-sem-ids=(priv,4096,deny)
project.max-msg-ids=(priv,4096,deny)
```

Кроме того, если Postgres Pro у вас выполняется внутри зоны, может понадобиться также увеличить лимиты на использование ресурсов зоны. Получить дополнительную информацию о `проектах` и команде `prctl` можно в *Руководстве системного администратора* (System Administrator's Guide), «Главе 2: Проекты и задачи» (Chapter2: Projects and Tasks).

17.4.2. RemoveIPC в systemd

Если используется systemd, необходимо позаботиться о том, чтобы ресурсы IPC (включая разделяемую память) не освобождались преждевременно операционной системой. Это особенно актуально при сборке и установке Postgres Pro из исходного кода. Пользователей дистрибутивных пакетов Postgres Pro это касается в меньшей степени, так как пользователь `postgres` обычно создаётся как системный пользователь.

Параметр `RemoveIPC` в `logind.conf` определяет, должны ли объекты IPC удаляться при полном выходе пользователя из системы. На системных пользователей это не распространяется. Этот параметр по умолчанию включён в стандартной сборке systemd, но в некоторых дистрибутивах операционных систем он по умолчанию отключён.

Обычно негативный эффект включения этого параметра проявляется в том, что объекты разделяемой памяти, используемые для параллельного выполнения запросов, удаляются без видимых причин, что приводит к появлению ошибок и предупреждений при попытке открыть и удалить их, например:

```
WARNING:  could not remove shared memory segment "/PostgreSQL.1450751626": No such file
 or directory
```

(ПРЕДУПРЕЖДЕНИЕ: ошибка при удалении сегмента разделяемой памяти "/PostgreSQL.1450751626": Нет такого файла или каталога) Различные типы объектов IPC (разделяемая память/семафоры, System V/POSIX) обрабатываются в systemd несколько по-разному, поэтому могут наблюдаться ситуации, когда некоторые ресурсы IPC не удаляются так, как другие. Однако полагаться на эти тонкие различия не рекомендуется.

Событие «выхода пользователя из системы» может произойти при выполнении задачи обслуживания или если администратор войдёт под именем `postgres`, а затем выйдет, либо случится что-то подобное, так что предотвратить это довольно сложно.

Какой пользователь является «системным», определяется во время компиляции systemd, исходя из значения `SYS_UID_MAX` в `/etc/login.defs`.

Скрипт упаковывания и развёртывания сервера должен предусмотрительно создавать пользователя `postgres` как системного пользователя, используя команды `useradd -r`, `adduser --system` или равнозначные.

Если же учётная запись пользователя была создана некорректно и изменить её невозможно, рекомендуется задать

```
RemoveIPC=no
```

в `/etc/systemd/logind.conf` или другом подходящем файле конфигурации.

Внимание

Необходимо предпринять минимум одно из этих двух действий, иначе сервер Postgres Pro будет очень нестабильным.

17.4.3. Ограничения ресурсов

В Unix-подобных операционных системах существуют различные типы ограничений ресурсов, которые могут влиять на работу сервера Postgres Pro. Особенно важны ограничения на число процессов для пользователя, число открытых файлов и объём памяти для каждого процесса. Каждое из этих ограничений имеет «жёсткий» и «мягкий» предел. Мягкий предел действительно ограничивает использование ресурса, но пользователь может увеличить его значение до жёсткого предела. Изменить жёсткий предел может только пользователь root. За изменение этих параметров отвечает системный вызов `setrlimit`. Управлять этими ресурсами в командной строке позволяет встроенная команда `ulimit` (в оболочках Bourne) и `limit` (csh). В системах семейства BSD различными ограничениями ресурсов, устанавливаемыми при входе пользователя, управляет файл `/etc/login.conf`. За подробностями обратитесь к документации операционной системы. Для Postgres Pro интерес представляют параметры `maxproc`, `openfiles` и `datasize`. Они могут задаваться, например так:

```
default:\
...
        :datasize-cur=256M:\
        :maxproc-cur=256:\
        :openfiles-cur=256:\
...
```

(Здесь `-cur` обозначает мягкий предел. Чтобы задать жёсткий предел, нужно заменить это окончание на `-max`.)

Ядро также может устанавливать общесистемные ограничения на использование некоторых ресурсов.

- В Linux максимальное число открытых файлов, которое поддерживает ядро, определяется параметром ядра `fs.file-max`. Изменить этот предел можно, воспользовавшись командой `sysctl -w fs.file-max=`*N*. Чтобы эти изменения сохранялись после перезагрузки, следует добавить присваивание в файл `/etc/sysctl.conf`. Максимальное число файлов для одного процесса задаётся при компиляции ядра; за дополнительными сведения обратитесь к `/usr/src/linux/Documentation/proc.txt`.

Сервер Postgres Pro использует для обслуживания каждого подключения отдельный процесс, так что возможное число процессов должно быть не меньше числа разрешённых соединений плюс число процессов, требуемых для остальной системы. Это обычно не проблема, но когда в одной системе работает множество серверов, предел может быть достигнут.

В качестве максимального числа открытых файлов по умолчанию обычно выбираются «социально-ориентированные» значения, позволяющие использовать одну систему нескольким пользователям так, чтобы ни один из них не потреблял слишком много системных ресурсов. Если вы запускаете в системе несколько серверов, это должно вполне устраивать, но на выделенных машинах может возникнуть желание увеличить этот предел.

С другой стороны, некоторые системы позволяют отдельным процессам открывать очень много файлов и если это делают сразу несколько процессов, они могут легко исчерпать общесистемный предел. Если вы столкнётесь с такой ситуацией, но не захотите менять общесистемное ограничение, вы можете ограничить использование открытых файлов сервером Postgres Pro, установив параметр конфигурации max_files_per_process.

Ещё одно ограничение в ядре, с которым можно столкнуться, когда устанавливается большое количество клиентских подключений, — максимальная длина очереди подключений к сокету.

Если количество запросов на подключение за короткий промежуток времени превышает этот максимум, некоторые из них будут отклонены до того, как главный процесс сможет их обработать, при этом клиенты получат неинформативное сообщение об ошибке подключения типа «Resource temporarily unavailable» (Ресурс временно недоступен) или «Connection refused» (Не удалось подключиться). Предел длины очереди на многих платформах по умолчанию составляет 128. Чтобы увеличить его, настройте соответствующий параметр ядра через sysctl и перезапустите главный процесс. Этот параметр называется `net.core.somaxconn` в Linux, `kern.ipc.soacceptqueue` в последних версиях FreeBSD и `kern.ipc.somaxconn` в macOS и на других платформах BSD.

17.4.4. Чрезмерное выделение памяти в Linux

В Linux 2.4 и новее механизм виртуальной памяти по умолчанию работает не оптимально для Postgres Pro. Вследствие того, что ядро выделяет память в чрезмерном объёме, оно может уничтожить главный управляющий процесс Postgres Pro (postmaster), если при выделении памяти процессу Postgres Pro или другому процессу виртуальная память будет исчерпана.

Когда это происходит, вы можете получить примерно такое сообщение ядра (где именно искать это сообщение, можно узнать в документации вашей системы):

```
Out of Memory: Killed process 12345 (postgres).
```

Это сообщение говорит о том, что процесс `postgres` был уничтожен из-за нехватки памяти. Хотя существующие подключения к базе данных будут работать по-прежнему, новые подключения приниматься не будут. Чтобы восстановить работу сервера, Postgres Pro придётся перезапустить.

Один из способов обойти эту проблему — запускать Postgres Pro на компьютере, где никакие другие процессы не займут всю память. Если физической памяти недостаточно, решить проблему также можно, увеличив объём пространства подкачки, так как уничтожение процессов при нехватке памяти происходит только когда заканчивается и физическая память, и место в пространстве подкачки.

Если памяти не хватает по вине самого Postgres Pro, эту проблему можно решить, изменив конфигурацию сервера. В некоторых случаях может помочь уменьшение конфигурационных параметров, связанных с памятью, а именно `shared_buffers` и `work_mem`. В других случаях проблема может возникать, потому что разрешено слишком много подключений к самому серверу баз данных. Чаще всего в такой ситуации стоит уменьшить число подключений `max_connections` и организовать внешний пул соединений.

В Linux 2.6 и новее «чрезмерное выделение» памяти можно предотвратить, изменив поведение ядра. Хотя при этом *OOM killer* (уничтожение процессов при нехватке памяти) всё равно может вызываться, вероятность такого уничтожения значительно уменьшается, а значит поведение системы становится более стабильным. Для этого нужно включить режим строгого выделения памяти, воспользовавшись `sysctl`:

```
sysctl -w vm.overcommit_memory=2
```

либо поместив соответствующую запись в `/etc/sysctl.conf`. Возможно, вы также захотите изменить связанный параметр `vm.overcommit_ratio`. За подробностями обратитесь к документации ядра *https://www.kernel.org/doc/Documentation/vm/overcommit-accounting*.

Другой подход, который можно применить (возможно, вместе с изменением `vm.overcommit_memory`), заключается в исключении процесса postmaster из числа возможных жертв при нехватке памяти. Для этого нужно задать для свойства *поправка очков OOM* этого процесса значение `-1000`. Проще всего это можно сделать, выполнив

```
echo -1000 > /proc/self/oom_score_adj
```

в скрипте запуска управляющего процесса непосредственно перед тем, как запускать postmaster. Заметьте, что делать это надо под именем root, иначе ничего не изменится; поэтому проще всего вставить эту команду в стартовый скрипт, принадлежащий пользователю root. Если вы делаете это, вы также должны установить в данном скрипте эти переменные окружения перед запуском главного процесса:

```
export PG_OOM_ADJUST_FILE=/proc/self/oom_score_adj
export PG_OOM_ADJUST_VALUE=0
```

С такими параметрами дочерние процессы главного будут запускаться с обычной, нулевой поправкой очков OOM, так что при необходимости механизм OOM сможет уничтожать их. Вы можете задать и другое значение для PG_OOM_ADJUST_VALUE, если хотите, чтобы дочерние процессы исполнялись с другой поправкой OOM. (PG_OOM_ADJUST_VALUE также можно опустить, в этом случае подразумевается нулевое значение.) Если вы не установите PG_OOM_ADJUST_FILE, дочерние процессы будут работать с той же поправкой очков OOM, которая задана для главного процесса, что неразумно, так всё это делается как раз для того, чтобы главный процесс оказался на особом положении.

В старых ядрах Linux /proc/self/oom_score_adj отсутствует, но та же функциональность может быть доступна через /proc/self/oom_adj. Эта переменная процесса работает так же, только значение, исключающее уничтожение процесса, равно -17, а не -1000.

> **Примечание**
>
> Некоторые дистрибутивы с ядрами Linux 2.4 содержат предварительную реализацию механизма sysctl overcommit, появившегося официально в 2.6. Однако если установить для vm.overcommit_memory значение 2 в ядре 2.4, ситуация не улучшится, а только ухудшится. Прежде чем модифицировать этот параметр в ядре 2.4, рекомендуется проанализировать исходный код вашего ядра (см. функцию vm_enough_memory в файле mm/mmap.c) и убедиться, что ядро поддерживает именно нужный вам режим. Наличие файла документации overcommit-accounting *не* следует считать признаком того, что он действительно поддерживается. В случае сомнений, обратитесь к эксперту по ядру или поставщику вашей системы.

17.4.5. Огромные страницы в Linux

Использование огромных страниц снижает накладные расходы при работе с большими непрерывными блоками памяти, что характерно для Postgres Pro, особенно при больших значениях shared_buffers. Чтобы можно было использовать эту возможность в Postgres Pro, ядро должно быть собрано с параметрами CONFIG_HUGETLBFS=y и CONFIG_HUGETLB_PAGE=y. Также вам понадобится настроить параметр ядра vm.nr_hugepages. Чтобы оценить требуемое количество огромных страниц, запустите Postgres Pro без поддержки огромных страниц и посмотрите на показатель VmPeak процесса postmaster, а также узнайте размер огромной страницы, воспользовавшись файловой системой /proc. Например, вы можете получить:

```
$ head -1 $PGDATA/postmaster.pid
4170
$ grep ^VmPeak /proc/4170/status
VmPeak:  6490428 kB
$ grep ^Hugepagesize /proc/meminfo
Hugepagesize:       2048 kB
```

В данном случае 6490428 / 2048 даёт примерно 3169.154, так что нам потребуется минимум 3170 огромных страниц, и мы можем задать это значение так:

```
$ sysctl -w vm.nr_hugepages=3170
```

Большее значение стоит указать, если огромные страницы будут использоваться и другими программами в этой системе. Не забудьте добавить этот параметр в /etc/sysctl.conf, чтобы он действовал и после перезагрузки.

Иногда ядро не может выделить запрошенное количество огромных страниц сразу, поэтому может потребоваться повторить эту команду или перезагрузить систему. (Немедленно после перезагрузки должен быть свободен больший объём памяти для преобразования в огромные страницы.) Чтобы проверить текущую ситуацию с размещением огромных страниц, выполните:

```
$ grep Huge /proc/meminfo
```

Также может потребоваться дать пользователю операционной системы, запускающему сервер БД, право использовать огромные страницы, установив его группу в `vm.hugetlb_shm_group` с помощью sysctl, и/или разрешить блокировать память, выполнив `ulimit -l`.

По умолчанию Postgres Pro использует огромные страницы, когда считает это возможным, а в противном случае переходит к обычным страницам. Чтобы задействовать огромные страницы принудительно, можно установить для huge_pages значение `on` в `postgresql.conf`. Заметьте, что с таким значением Postgres Pro не сможет запуститься, если не получит достаточного количества огромных страниц.

Более подробно о механизме огромных страниц в Linux можно узнать в документации ядра: *https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt*.

17.5. Выключение сервера

Сервер баз данных можно отключить несколькими способами. Вы выбираете тот или иной вариант отключения, посылая разные сигналы главному процессу `postgres`.

SIGTERM

Запускает так называемое *умное выключение*. Получив SIGTERM, сервер перестаёт принимать новые подключения, но позволяет всем существующим сеансам закончить работу в штатном режиме. Сервер будет отключён только после завершения всех сеансов. Если сервер находится в режиме архивации, сервер дополнительно ожидает выхода из этого режима. При этом в данном случае сервер позволяет устанавливать новые подключения, но только для суперпользователей (это исключение позволяет суперпользователю подключиться и прервать архивацию). Если получая этот сигнал, сервер находится в процессе восстановления, восстановление и потоковая репликация будут прерваны только после завершения всех обычных сеансов.

SIGINT

Запускает *быстрое выключение*. Сервер запрещает новые подключения и посылает всем работающим серверным процессам сигнал SIGTERM, в результате чего их транзакции прерываются и сами процессы завершаются. Управляющий процесс ждёт, пока будут завершены все эти процессы и затем завершается сам. Если сервер находится в режиме архивации, архивация прерывается, так что архив оказывается неполным.

SIGQUIT

Запускает *немедленное выключение*. Сервер отправляет всем дочерним процессам сигнал SIGQUIT и ждёт их завершения. Если какие-либо из них не завершаются в течение 5 секунд, им посылается SIGKILL. Главный процесс сервера завершается, как только будут завершены все дочерние процессы, не выполняя обычную процедуру остановки БД. В результате при последующем запуске будет запущен процесс восстановления (воспроизведения изменений из журнала). Такой вариант выключения рекомендуется только в экстренных ситуациях.

Удобную возможность отправлять эти сигналы, отключающие сервер, предоставляет программа pg_ctl. Кроме того, в системах, отличных от Windows, соответствующий сигнал можно отправить с помощью команды `kill`. PID основного процесса `postgres` можно узнать, воспользовавшись программой `ps`, либо прочитав файл `postmaster.pid` в каталоге данных. Например, можно выполнить быстрое выключение так:

```
$ kill -INT `head -1 /usr/local/pgsql/data/postmaster.pid`
```

> **Важно**
>
> Для выключения сервера не следует использовать сигнал SIGKILL. При таком выключении сервер не сможет освободить разделяемую память и семафоры, и, возможно, это придётся

делать вручную, чтобы сервер мог запуститься снова. Кроме того, при уничтожении главного процесса `postgres` сигналом SIGKILL, он не успеет передать этот сигнал своим дочерним процессам, так что может потребоваться завершать и их вручную.

Чтобы завершить отдельный сеанс, не прерывая работу других сеансов, воспользуйтесь функцией `pg_terminate_backend()` (см. Таблицу 9.78) или отправьте сигнал SIGTERM дочернему процессу, обслуживающему этот сеанс.

17.6. Обновление кластера Postgres Pro

В этом разделе рассказывается, как обновить ваш кластер базы данных с одной версии Postgres Pro на другую.

Текущие номера версий Postgres Pro состоят из номеров основной и корректирующей (дополнительной) версии. Например, в номере версии 10.1 число 10 обозначает основную версию, а 1 — дополнительную. Для выпусков Postgres Pro до версии 10.0 номера состояли из трёх чисел (например, 9.5.3). Тогда основная версия образовывалась группой их двух чисел (например, 9.5), а дополнительная задавалась третьим числом (например, 3, что означало, что это третий корректирующий выпуск основной версии 9.5).

В корректирующих выпусках никогда не меняется внутренний формат хранения, и они всегда совместимы с предыдущими и последующими выпусками той же основной версии. Например, версия 10.1 совместима с версией 10.0 и версией 10.6. Подобным образом, например, версия 9.5.3 совместима с 9.5.0, 9.5.1 и 9.5.6. Для обновления версии на совместимую достаточно просто заменить исполняемые файлы при выключенном сервере и затем запустить сервер. Каталог данных при этом не затрагивается, так что обновить корректирующую версию довольно просто.

При обновлении *основных* версий Postgres Pro внутренний формат данных может меняться, что усложняет процедуру обновления. Традиционный способ переноса данных в новую основную версию — выгрузить данные из старой версии, а затем загрузить их в новую (это не самый быстрый вариант). Выполнить обновление быстрее позволяет pg_upgrade. Также для обновления можно использовать репликацию, как описано ниже.

Изменения основной версии обычно приносят какие-либо видимые пользователю несовместимости, которые могут требовать доработки приложений. Все подобные изменения описываются в замечаниях к выпуску (Приложение E); обращайте особое внимание на раздел «Migration» (Миграция). Хотя вы можете перейти с одной основной версии на другую, пропустив промежуточные версии, обязательно ознакомьтесь с замечаниями к каждому выпуску, в том числе для каждой пропускаемой версии.

Осторожные пользователи обычно тестируют свои клиентские приложения с новой версией, прежде чем переходить на неё полностью; поэтому часто имеет смысл установить рядом старую и новую версии. При тестировании обновления основной версии Postgres Pro изучите следующие области возможных изменений:

Администрирование

Средства и функции, предоставляемые администраторам для наблюдения и управления сервером, часто меняются и совершенствуются в каждой новой версии.

SQL

В этой области чаще наблюдается появление новых возможностей команд SQL, чем изменение поведения существующих, если только об этом не говорится в замечаниях к выпуску.

API библиотек

Обычно библиотеки типа libpq только расширяют свою функциональность, если об обратном так же не говорится в замечаниях к выпуску.

Системные каталоги

Изменения в системных каталогах обычно влияют только на средства управления базами данных.

API сервера для кода на C

Сюда относятся изменения в API серверных функций, которые написаны на языке программирования C. Такие изменения затрагивают код, обращающийся к служебным функциям глубоко внутри сервера.

17.6.1. Обновление данных с применением pg_dumpall

Один из вариантов обновления заключается в выгрузке данных из одной основной версии Postgres Pro и загрузке их в другую — для этого необходимо использовать средство *логического* копирования, например pg_dumpall; копирование на уровне файловой системы не подходит. (В самом сервере есть проверки, которые не дадут использовать каталог данных от несовместимой версии Postgres Pro, так что если попытаться запустить с существующим каталогом данных неправильную версию сервера, никакого вреда не будет.)

Для создания копии рекомендуется применять программы pg_dump и pg_dumpall от *новой* версии Postgres Pro, чтобы воспользоваться улучшенными функциями, которые могли в них появиться. Текущие версии этих программ способны читать данные любой версии сервера, начиная с 8.0.

В следующих указаниях предполагается, что сервер установлен в каталоге `/usr/local/pgsql`, а данные находятся в `/usr/local/pgsql/data`. Вам нужно подставить свои пути.

1. При запуске резервного копирования убедитесь в том, что в базе данных не производятся изменения. Изменения не повлияют на целостность полученной копии, но изменённые данные, само собой, в неё не попадут. Если потребуется, измените разрешения в файле `/usr/local/pgsql/data/pg_hba.conf` (или подобном), чтобы подключиться к серверу могли только вы. За дополнительными сведениями об управлении доступом обратитесь к Главе 19.

   Чтобы получить копию всех ваших данных, введите:

   ```
   pg_dumpall > выходной_файл
   ```

   Для создания резервной копии вы можете воспользоваться программой pg_dumpall от текущей версии сервера; за подробностями обратитесь к Подразделу 24.1.2. Однако для лучшего результата стоит попробовать pg_dumpall из Postgres Pro Standard 10.23.1, так как в эту версию вошли исправления ошибок и усовершенствования, по сравнению с предыдущими версиями. Хотя этот совет может показаться абсурдным, ведь новая версия ещё не установлена, ему стоит последовать, если вы планируете установить новую версию рядом со старой. В этом случае вы сможете выполнить установку как обычно, а перенести данные позже. Это также сократит время обновления.

2. Остановите старый сервер:

   ```
   pg_ctl stop
   ```

   В системах, где Postgres Pro запускается при загрузке, должен быть скрипт запуска, с которым можно сделать то же самое. Например, в Red Hat Linux может сработать такой вариант:

   ```
   /etc/rc.d/init.d/postgresql stop
   ```

   Подробнее запуск и остановка сервера описаны в Главе 17.

3. При восстановлении из резервной копии удалите или переименуйте старый каталог, где был установлен сервер, если его имя не привязано к версии. Разумнее будет переименовать каталог, а не удалять его, чтобы его можно было восстановить в случае проблем. Однако учтите, что этот каталог может занимать много места на диске. Переименовать каталог можно, например так:

   ```
   mv /usr/local/pgsql /usr/local/pgsql.old
   ```

(Этот каталог нужно переименовывать (перемещать) как единое целое, чтобы относительные пути в нём не изменились.)

4. Установите новую версию Postgres Pro Standard.
5. При необходимости создайте новый кластер баз данных. Помните, что следующие команды нужно выполнять под именем специального пользователя БД (вы уже действуете под этим именем, если производите обновление).

   /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
6. Перенесите изменения, внесённые в предыдущие версии `pg_hba.conf` и `postgresql.conf`.
7. Запустите сервер баз данных, так же применяя учётную запись специального пользователя БД:

   /usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
8. Наконец, восстановите данные из резервной копии, выполнив:

   /usr/local/pgsql/bin/psql -d postgres -f *выходной_файл*

   (При этом будет использоваться *новый* psql.)

Минимизировать время отключения сервера можно, установив новый сервер в другой каталог и запустив параллельно оба сервера, старый и новый, с разными портами. Затем можно будет перенести данные примерно так:

```
pg_dumpall -p 5432 | psql -d postgres -p 5433
```

17.6.2. Обновление данных с применением pg_upgrade

Модуль pg_upgrade позволяет обновить инсталляцию Postgres Pro с одной основной версии на другую непосредственно на месте. Такое обновление может выполняться за считанные минуты, особенно в режиме `--link`. Для него требуются примерно те же подготовительные действия, что и для варианта с pg_dumpall: запустить/остановить сервер, выполнить initdb. Все эти действия описаны в документации pg_upgrade.

17.6.3. Обновление данных с применением репликации

Также можно использовать некоторые методы репликации, например Slony, для создания резервного сервера с обновлённой версией Postgres Pro. Это возможно благодаря тому, что Slony поддерживает репликацию между разными основными версиями Postgres Pro. Резервный сервер может располагаться как на том же компьютере, так и на другом. Как только синхронизация с главным сервером (где работает старая версия Postgres Pro) будет завершена, можно сделать главным новый сервер, а старый экземпляр базы данных просто отключить. При таком переключении обновление можно осуществить, прервав работу сервера всего на несколько секунд.

17.7. Защита от подмены сервера

Когда сервер работает, злонамеренный пользователь не может подставить свой сервер вместо него. Однако если сервер отключён, локальный пользователь может подменить нормальный сервер, запустив свой собственный. Поддельный сервер сможет читать пароли и запросы клиентов, хотя не сможет вернуть никакие данные, так как каталог `PGDATA` будет защищён от чтения посторонними пользователями. Такая подмена возможна потому, что любой пользователь может запустить сервер баз данных; клиент, со своей стороны, не может обнаружить подмену, если его не настроить дополнительно.

Один из способов предотвратить подмену для `локальных` подключений — использовать каталог Unix-сокетов (unix_socket_directories), в который сможет писать только проверенный локальный пользователь. Это не позволит злонамеренному пользователю создать в этом каталоге свой файл сокета. Если вас беспокоит, что некоторые приложения при этом могут обращаться к файлу сокета в `/tmp` и, таким образом, всё же будут уязвимыми, создайте при загрузке операционной системы символическую ссылку `/tmp/.s.PGSQL.5432`, указывающую на перемещённый файл

сокета. Возможно, вам также придётся изменить скрипт очистки `/tmp`, чтобы он не удалял эту ссылку.

Также клиенты могут защитить `локальные` подключения, установив в параметре `requirepeer` имя пользователя, который должен владеть серверным процессом, подключённым к сокету.

Лучший способ защиты от подмены для соединений TCP — использовать сертификаты SSL и добиться, чтобы клиенты проверяли сертификат сервера. Для этого надо настроить сервер, чтобы он принимал только подключения `hostssl` (см. Раздел 19.1) и имел ключ и сертификаты SSL (см. Раздел 17.9). Тогда TCP-клиент должен будет подключаться к серверу с параметром `sslmode=verify-ca` или `verify-full` и у него должен быть установлен соответствующий корневой сертификат (см. Подраздел 31.18.1).

17.8. Возможности шифрования

Postgres Pro обеспечивает шифрование на разных уровнях и даёт гибкость в выборе средств защиты данных в случае кражи сервера, от недобросовестных администраторов или в небезопасных сетях. Шифрование может также требоваться для защиты конфиденциальных данных, например, медицинских сведений или финансовых транзакций.

Шифрование паролей

Пароли пользователей базы данных хранятся в виде хешей (алгоритм хеширования определяется параметром password_encryption), так что администратор не может узнать, какой именно пароль имеет пользователь. Если шифрование SCRAM или MD5 применяется и при проверке подлинности, пароль не присутствует на сервере в открытом виде даже кратковременно, так как клиент шифрует его перед тем как передавать по сети. Предпочтительным методом является SCRAM, так как это стандарт, принятый в Интернете, и он более безопасен, чем собственный протокол проверки MD5 в PostgreSQL.

Шифрование избранных столбцов

Модуль pgcrypto позволяет хранить в зашифрованном виде избранные поля. Это полезно, если ценность представляют только некоторые данные. Чтобы прочитать эти поля, клиент передаёт дешифрующий ключ, сервер расшифровывает данные и выдаёт их клиенту.

Расшифрованные данные и ключ дешифрования находятся на сервере в процессе расшифровывания и передачи данных. Именно в этот момент данные и ключи могут быть перехвачены тем, кто имеет полный доступ к серверу баз данных, например, системным администратором.

Шифрование раздела данных

Шифрование хранилища данных можно реализовать на уровне файловой системы или на уровне блоков. В Linux можно воспользоваться шифрованными файловыми системами eCryptfs и EncFS, а во FreeBSD есть PEFS. Шифрование всего диска на блочном уровне в Linux можно организовать, используя dm-crypt + LUKS, а во FreeBSD — модули GEOM, geli и gbde. Подобные возможности есть и во многих других операционных системах, включая Windows.

Этот механизм не позволяет читать незашифрованные данные с дисков в случае кражи дисков или всего компьютера. При этом он не защищает данные от чтения, когда эта файловая система смонтирована, так как на смонтированном устройстве операционная система видит все данные в незашифрованном виде. Однако чтобы смонтировать файловую систему, нужно передать операционной системе ключ (иногда он хранится где-то на компьютере, который выполняет монтирование).

Шифрование данных при передаче по сети

SSL-соединения шифруют все данные, передаваемые по сети: пароль, запросы и возвращаемые данные. Файл `pg_hba.conf` позволяет администраторам указать, для каких узлов будут разрешены незашифрованные соединения (`host`), а для каких будет требоваться SSL (`hostssl`).

Кроме того, и на стороне клиента можно разрешить подключения к серверам только с SSL. Для шифрования трафика также можно применять stunnel и SSH.

Проверка подлинности сервера SSL

И клиент, и сервер могут проверять подлинность друг друга по сертификатам SSL. Это требует дополнительной настройки на каждой стороне, но даёт более надёжную гарантию подлинности, чем обычные пароли. С такой защитой подставной компьютер не сможет представлять из себя сервер с целью получить пароли клиентов. Она также предотвращает атаки с посредником («man in the middle»), когда компьютер между клиентом и сервером представляется сервером и незаметно передаёт все запросы и данные между клиентом и подлинным сервером.

Шифрование на стороне клиента

Если системный администратор сервера, где работает база данных, не является доверенным, клиент должен сам шифровать данные; тогда незашифрованные данные никогда не появятся на этом сервере. В этом случае клиент шифрует данные, прежде чем передавать их серверу, а получив из базы данных результаты, он расшифровывает их для использования.

17.9. Защита соединений TCP/IP с применением SSL

В Postgres Pro встроена поддержка SSL для шифрования трафика между клиентом и сервером, что повышает уровень безопасности системы.

Когда в установленном сервере Postgres Pro разрешена поддержка SSL, его можно запустить с включённым механизмом SSL, задав в `postgresql.conf` для параметра ssl значение `on`. Запущенный сервер будет принимать как обычные, так и SSL-подключения в одном порту TCP и будет согласовывать использование SSL с каждым клиентом. По умолчанию клиент выбирает режим подключения сам; как настроить сервер, чтобы он требовал использования только SSL для всех или некоторых подключений, вы можете узнать в Разделе 19.1.

Postgres Pro читает системный файл конфигурации OpenSSL. По умолчанию этот файл называется `openssl.cnf` и находится в каталоге, который сообщает команда `openssl version -d`. Если требуется указать другое расположение файла конфигурации, его можно задать в переменной окружения `OPENSSL_CONF`.

OpenSSL предоставляет широкий выбор шифров и алгоритмов аутентификации разной защищённости. Хотя список шифров может быть задан непосредственно в файле конфигурации OpenSSL, можно задать отдельные шифры именно для сервера баз данных, указав их в параметре ssl_ciphers в `postgresql.conf`.

> **Примечание**
>
> Накладные расходы, связанные с шифрованием, в принципе можно исключить, ограничившись только проверкой подлинности, то есть применяя шифр `NULL-SHA` или `NULL-MD5`. Однако в этом случае посредник сможет пропускать через себя и читать весь трафик между клиентом и сервером. Кроме того, шифрование привносит минимальную дополнительную нагрузку по сравнению с проверкой подлинности. По этим причинам использовать шифры NULL не рекомендуется.

Чтобы сервер мог работать в режиме SSL, ему необходимы файлы с сертификатом сервера и закрытым ключом. По умолчанию это должны быть файлы `server.crt` и `server.key`, соответственно, расположенные в каталоге данных, но можно использовать и другие имена и местоположения файлов, задав их в конфигурационных параметрах ssl_cert_file и ssl_key_file.

В Unix-подобных системах к файлу `server.key` должен быть запрещён любой доступ группы и всех остальных; чтобы установить такое ограничение, выполните `chmod 0600 server.key`. Возможен и

другой вариант, когда этим файлом владеет root, а группа имеет доступ на чтение (то есть, маска разрешений `0640`). Данный вариант предназначен для систем, в которых файлами сертификатов и ключей управляет сама операционная система. В этом случае пользователь, запускающий сервер Postgres Pro, должен быть членом группы, имеющей доступ к указанным файлам сертификата и ключа.

Если закрытый ключ защищён паролем, сервер спросит этот пароль и не будет запускаться, пока он не будет введён. Использование такого пароля лишает возможности изменять конфигурацию SSL без перезагрузки сервера. Более того, закрытые ключи, защищённые паролем, не годятся для использования в Windows.

Первым сертификатом в `server.crt` должен быть сертификат сервера, так как он должен соответствовать закрытому ключу сервера. В этот файл также могут быть добавлены сертификаты «промежуточных» центров сертификации. Это избавляет от необходимости хранить все промежуточные сертификаты на клиентах, при условии, что корневой и промежуточные сертификаты были созданы с расширениями `v3_ca`. (При этом в основных ограничениях сертификата устанавливается свойство `CA`, равное `true`.) Это также упрощает управление промежуточными сертификатами с истекающим сроком.

Добавлять корневой сертификат в `server.crt` нет необходимости. Вместо этого клиенты должны иметь этот сертификат в цепочке сертификатов сервера.

17.9.1. Использование клиентских сертификатов

Чтобы клиенты должны были предоставлять серверу доверенные сертификаты, поместите сертификаты корневых центров сертификации (ЦС), которым вы доверяете, в файл в каталоге данных, укажите в параметре ssl_ca_file в `postgresql.conf` имя этого файла и добавьте параметр аутентификации `clientcert=1` в соответствующие строки `hostssl` в `pg_hba.conf`. В результате от клиента в процессе установления SSL-подключения будет затребован сертификат. (Как настроить сертификаты на стороне клиента, описывается в Разделе 31.18.) Получив сертификат, сервер будет проверять, подписан ли этот сертификат одним из доверенным центром сертификации.

Промежуточные сертификаты, которые составляют цепочку с существующими корневыми сертификатами, можно также включить в файл `root.crt`, если вы не хотите хранить их на стороне клиента (предполагается, что корневой и промежуточный сертификаты были созданы с расширениями `v3_ca`). Если установлен параметр ssl_crl_file, также проверяются списки отзыва сертификатов (Certificate Revocation List, CRL).

Параметр аутентификации `clientcert` можно использовать с любым методом проверки подлинности, но только в строках `pg_hba.conf` типа `hostssl`. Когда `clientcert` не задан или равен 0, сервер всё же будет проверять все представленные клиентские сертификаты по своему списку ЦС (если он настроен), но позволит подключаться клиентам без сертификата.

Если вы используете клиентские сертификаты, вы можете также применить метод аутентификации `cert`, чтобы сертификаты обеспечивали не только защиту соединений, но и проверку подлинности пользователей. За подробностями обратитесь к Подразделу 19.3.9. (Устанавливать `clientcert=1` явно при использовании метода аутентификации `cert` не требуется.)

17.9.2. Файлы, используемые SSL-сервером

В Таблице 17.2 кратко описаны все файлы, имеющие отношение к настройке SSL на сервере. (Здесь приведены стандартные или типичные имена файлов. В конкретной системе они могут быть другими.)

Таблица 17.2. Файлы, используемые SSL-сервером

Файл	Содержимое	Назначение
ssl_cert_file (`$PGDATA/server.crt`)	сертификат сервера	отправляется клиенту для идентификации сервера

Файл	Содержимое	Назначение
ssl_key_file (`$PGDATA/server.key`)	закрытый ключ сервера	подтверждает, что сертификат сервера был передан его владельцем; не гарантирует, что его владельцу можно доверять
ssl_ca_file (`$PGDATA/root.crt`)	сертификаты доверенных ЦС	позволяет проверить, что сертификат клиента подписан доверенным центром сертификации
ssl_crl_file (`$PGDATA/root.crl`)	сертификаты, отозванные центрами сертификации	сертификат клиента должен отсутствовать в этом списке

Сервер читает эти файлы при запуске или при перезагрузке конфигурации. В системах Windows они также считываются заново, когда для нового клиентского подключения запускается новый обслуживающий процесс.

Если в этих файлах при запуске сервера обнаружится ошибка, сервер откажется запускаться. Но если ошибка обнаруживается при перезагрузке конфигурации, эти файлы игнорируются и продолжает использоваться старая конфигурация SSL. В системах Windows, если в одном из этих файлов обнаруживается ошибка при запуске обслуживающего процесса, этот процесс не сможет устанавливать SSL-соединения. Во всех таких случаях в журнал событий сервера выводится сообщение об ошибке.

17.9.3. Создание сертификатов

Чтобы создать простой самоподписанный сертификат для сервера, действующий 365 дней, выполните следующую команду OpenSSL, заменив *dbhost.yourdomain.com* именем компьютера, где размещён сервер:

```
openssl req -new -x509 -days 365 -nodes -text -out server.crt \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
```

Затем выполните:

```
chmod og-rwx server.key
```

так как сервер не примет этот файл, если разрешения будут более либеральными, чем показанные. За дополнительными сведениями относительно создания закрытого ключа и сертификата сервера обратитесь к документации OpenSSL.

Хотя самоподписанный сертификат может успешно применяться при тестировании, в производственной среде следует использовать сертификат, подписанный центром сертификации (ЦС) (обычно это корневой ЦС предприятия).

Чтобы создать сертификат сервера, подлинность которого смогут проверять клиенты, сначала создайте запрос на получение сертификата (CSR) и файлы открытого/закрытого ключа:

```
openssl req -new -nodes -text -out root.csr \
  -keyout root.key -subj "/CN=root.yourdomain.com"
chmod og-rwx root.key
```

Затем подпишите запрос ключом, чтобы создать корневой центр сертификации (с файлом конфигурации OpenSSL, помещённым в Linux в расположение по умолчанию):

```
openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt
```

Наконец, создайте сертификат сервера, подписанный новым корневым центром сертификации:

```
openssl req -new -nodes -text -out server.csr \
```

```
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key

openssl x509 -req -in server.csr -text -days 365 \
  -CA root.crt -CAkey root.key -CAcreateserial \
  -out server.crt
```

`server.crt` и `server.key` должны быть сохранены на сервере, а `root.crt` — на клиенте, чтобы клиент мог убедиться в том, что конечный сертификат сервера подписан центром сертификации, которому он доверяет. Файл `root.key` следует хранить в изолированном месте для создания сертификатов в будущем.

Также возможно создать цепочку доверия, включающую промежуточные сертификаты:

```
# корневой сертификат
openssl req -new -nodes -text -out root.csr \
  -keyout root.key -subj "/CN=root.yourdomain.com"
chmod og-rwx root.key
openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt

# промежуточный
openssl req -new -nodes -text -out intermediate.csr \
  -keyout intermediate.key -subj "/CN=intermediate.yourdomain.com"
chmod og-rwx intermediate.key
openssl x509 -req -in intermediate.csr -text -days 1825 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -CA root.crt -CAkey root.key -CAcreateserial \
  -out intermediate.crt

# конечный
openssl req -new -nodes -text -out server.csr \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key
openssl x509 -req -in server.csr -text -days 365 \
  -CA intermediate.crt -CAkey intermediate.key -CAcreateserial \
  -out server.crt
```

`server.crt` и `intermediate.crt` следует сложить вместе в пакет сертификатов и сохранить на сервере. Также на сервере следует сохранить `server.key`. Файл `root.crt` нужно сохранить на клиенте, чтобы клиент мог убедиться в том, что конечный сертификат сервера был подписан по цепочке сертификатов, связанных с корневым сертификатом, которому он доверяет. Файлы `root.key` и `intermediate.key` следует хранить в изолированном месте для создания сертификатов в будущем.

17.10. Защита соединений TCP/IP с применением туннелей SSH

Для защиты сетевых соединений клиентов с сервером Postgres Pro можно применить SSH. При правильном подходе это позволяет обеспечить должный уровень защиты сетевого трафика, даже для клиентов, не поддерживающих SSL.

Прежде всего убедитесь, что на компьютере с сервером Postgres Pro также работает сервер SSH и вы можете подключиться к нему через `ssh` каким-нибудь пользователем; затем вы сможете установить защищённый туннель с этим удалённым сервером. Защищённый туннель прослушивает локальный порт и перенаправляет весь трафик в порт удалённого компьютера. Трафик, передаваемый в удалённый порт, может поступить на его адрес `localhost` или на другой привязанный адрес, если это требуется. При этом не будет видно, что трафик поступает с

вашего компьютера. Следующая команда устанавливает защищённый туннель между клиентским компьютером и удалённой машиной `foo.com`:

```
ssh -L 63333:localhost:5432 joe@foo.com
```

Первое число в аргументе `-L`, 63333 — это локальный номер порта туннеля; это может быть любой незанятый порт. (IANA резервирует порты с 49152 по 65535 для частного использования.) Имя или IP-адрес после него обозначает привязанный адрес с удалённой стороны, к которому вы подключаетесь, в данном случае это `localhost` (и это же значение по умолчанию). Второе число, 5432 — порт с удалённой стороны туннеля, то есть порт вашего сервера баз данных. Чтобы подключиться к этому серверу через созданный тоннель, нужно подключиться к порту 63333 на локальном компьютере:

```
psql -h localhost -p 63333 postgres
```

Для сервера баз данных это будет выглядеть так, как будто вы пользователь `joe` компьютера `foo.com`, подключающийся к адресу `localhost`, и он будет применять ту процедуру проверки подлинности, которая установлена для подключений данного пользователя к этому привязанному адресу. Заметьте, что сервер не будет считать такое соединение защищённым SSL, так как на самом деле трафик между сервером SSH и сервером Postgres Pro не защищён. Это не должно нести какие-то дополнительные риски, так как эти серверы работают на одном компьютере.

Чтобы настроенный таким образом туннель работал, вы должны иметь возможность подключаться к компьютеру через `ssh` под именем `joe@foo.com`, так же, как это происходит при установлении терминального подключения с помощью `ssh`.

Вы также можете настроить перенаправление портов примерно так:

```
ssh -L 63333:foo.com:5432 joe@foo.com
```

Но в этом случае с точки зрения сервера подключение будет приходить на его сетевой адрес `foo.com`, а он по умолчанию не прослушивается (вследствие указания `listen_addresses = 'localhost'`). Обычно требуется другое поведение.

Если вам нужно «перейти» к серверу баз данных через некоторый шлюз, это можно организовать примерно так:

```
ssh -L 63333:db.foo.com:5432 joe@shell.foo.com
```

Заметьте, что в этом случае трафик между `shell.foo.com` и `db.foo.com` не будет защищён туннелем SSH. SSH предлагает довольно много вариантов конфигурации, что позволяет организовывать защиту сети разными способами. За подробностями обратитесь к документации SSH.

> **Подсказка**
>
> Существуют и другие приложения, которые могут создавать безопасные туннели, применяя по сути тот же подход, что был описан выше.

17.11. Регистрация журнала событий в Windows

Чтобы зарегистрировать библиотеку журнала событий в Windows, выполните следующую команду:

regsvr32 *каталог_библиотек_pgsql*/pgevent.dll

При этом в реестре будут созданы необходимые записи для средства просмотра событий, относящиеся к источнику событий по умолчанию с именем `Postgres Pro`.

Чтобы задать другое имя источника событий (см. event_source), укажите ключи `/n` и `/i`:

regsvr32 /n /i:*имя_источника_событий каталог_библиотек_pgsql*/pgevent.dll

Чтобы разрегистрировать библиотеку журнала событий в операционной системе, выполните команду:

regsvr32 /u [/i:*имя_источника_событий*] *каталог_библиотек_pgsql*/pgevent.dll

> **Примечание**
>
> Чтобы сервер баз данных записывал сообщения в журнал событий, добавьте `eventlog` в параметр log_destination в `postgresql.conf`.

Глава 18. Настройка сервера

На работу системы баз данных оказывают влияние множество параметров конфигурации. В первом разделе этой главы рассказывается, как управлять этими параметрами, а в последующих разделах подробно описывается каждый из них.

18.1. Изменение параметров

18.1.1. Имена и значения параметров

Имена всех параметров являются регистронезависимыми. Каждый параметр принимает значение одного из пяти типов: логический, строка, целое, число с плавающей точкой или перечисление. От типа значения зависит синтаксис установки этого параметра:

- *Логический:* Значения могут задаваться строками `on`, `off`, `true`, `false`, `yes`, `no`, `1`, `0` (регистр не имеет значения), либо как достаточно однозначный префикс одной из этих строк.
- *Строка:* Обычно строковое значение заключается в апострофы (при этом внутренние апострофы дублируются). Однако если значение является простым числом или идентификатором, апострофы обычно можно опустить.
- *Число (целое или с плавающей точкой):* Десятичная точка как разделитель целой и дробной части допускается только для параметров, принимающих числа с плавающей точкой. Разделители разрядов в записи числа не принимаются. Заключать в апострофы число не требуется.
- *Число с единицей измерения:* Некоторые числовые параметры задаются с единицами измерения, так как они описывают количества информации или времени. Единицами могут быть килобайты, блоки (обычно восемь килобайт), миллисекунды, секунды или минуты. При указании только числового значения для такого параметра единицей измерения будет считаться единица по умолчанию для параметра, которая указывается в `pg_settings.unit`. Для удобства параметры также можно задавать, указывая единицу измерения явно, например, задать `'120 ms'` для значения времени, при этом такое значение будет переведено в основную единицу измерения параметра. Заметьте, что для этого значение должно записываться в виде строки (в апострофах). Имя единицы является регистронезависимым, а между ним и числом допускаются пробельные символы.
  - Допустимые единицы информации: `kB` (килобайты), `MB` (мегабайты), `GB` (гигабайты) и `TB` (терабайты). Множителем единиц информации считается 1024, не 1000.
  - Допустимые единицы времени: `ms` (миллисекунды), `s` (секунды), `min` (минуты), `h` (часы) и `d` (дни).
- *Перечисление:* Параметры, имеющие тип перечисление, записываются так же, как строковые параметры, но могут иметь только ограниченный набор значений. Список допустимых значений такого параметра задаётся в `pg_settings.enumvals`. В значениях перечислений регистр не учитывается.

18.1.2. Определение параметров в файле конфигурации

Самый основной способ установки этих параметров — определение их значений в файле `postgresql.conf`, который обычно находится в каталоге данных. При инициализации каталога кластера БД в этот каталог помещается копия стандартного файла. Например, он может выглядеть так:

```
# Это комментарий
log_connections = yes
log_destination = 'syslog'
search_path = '"$user", public'
shared_buffers = 128MB
```

Каждый параметр определяется в отдельной строке. Знак равенства в ней между именем и значением является необязательным. Пробельные символы в строке не играют роли (кроме значений, заключённых в апострофы), а пустые строки игнорируются. Знаки решётки (#) обозначают продолжение строки как комментарий. Значения параметров, не являющиеся простыми идентификаторами или числами, должны заключаться в апострофы. Чтобы включить в такое значение собственно апостроф, его следует продублировать (предпочтительнее) или предварить обратной косой чертой. Если один и тот же параметр определяется в файле конфигурации неоднократно, действовать будет только последнее определение, остальные игнорируются.

Параметры, установленные таким образом, задают значения по умолчанию для данного кластера. Эти значения будут действовать в активных сеансах, если не будут переопределены. В следующих разделах описывается, как их может переопределить администратор или пользователь.

Основной процесс сервера перечитывает файл конфигурации заново, получая сигнал SIGHUP; послать его проще всего можно, запустив `pg_ctl reload` в командной строке или вызвав SQL-функцию `pg_reload_conf()`. Основной процесс сервера передаёт этот сигнал всем остальным запущенным серверным процессам, так что существующие сеансы тоже получают новые значения (после того, как завершится выполнение текущей команды клиента). Также возможно послать этот сигнал напрямую одному из серверных процессов. Учтите, что некоторые параметры можно установить только при запуске сервера; любые изменения их значений в файле конфигурации не будут учитываться до перезапуска сервера. Более того, при обработке SIGHUP игнорируются неверные значения параметров (но об этом сообщается в журнале).

В дополнение к `postgresql.conf` в каталоге данных Postgres Pro содержится файл `postgresql.auto.conf`, который имеет тот же формат, что и `postgresql.conf`, но предназначен для автоматического изменения, а не для редактирования вручную. Этот файл содержит параметры, задаваемые командой ALTER SYSTEM. Он считывается одновременно с `postgresql.conf` и заданные в нём параметры действуют таким же образом. Параметры в `postgresql.auto.conf` переопределяют те, что указаны в `postgresql.conf`.

Вносить изменения в `postgresql.auto.conf` можно и с использованием внешних средств. Однако это не рекомендуется делать в процессе работы сервера, так эти изменения могут быть потеряны при параллельном выполнении команды `ALTER SYSTEM`. Внешние программы могут просто добавлять новые определения параметров в конец файла или удалять повторяющиеся определения и/или комментарии (как делает `ALTER SYSTEM`).

Системное представление `pg_file_settings` может быть полезным для предварительной проверки изменений в файлах конфигурации или для диагностики проблем, если сигнал SIGHUP не даёт желаемого эффекта.

18.1.3. Управление параметрами через SQL

В Postgres Pro есть три SQL-команды, задающие для параметров значения по умолчанию. Уже упомянутая команда ALTER SYSTEM даёт возможность изменять глобальные значения средствами SQL; она функционально равнозначна редактированию `postgresql.conf`. Кроме того, есть ещё две команды, которые позволяют задавать значения по умолчанию на уровне баз данных и ролей:

- Команда ALTER DATABASE позволяет переопределить глобальные параметры на уровне базы данных.
- Команда ALTER ROLE позволяет переопределить для конкретного пользователя как глобальные, так и локальные для базы данных параметры.

Значения, установленные командами `ALTER DATABASE` и `ALTER ROLE`, применяются только при новом подключении к базе данных. Они переопределяют значения, полученные из файлов конфигурации или командной строки сервера, и применяются по умолчанию в рамках сеанса. Заметьте, что некоторые параметры невозможно изменить после запуска сервера, поэтому их нельзя установить этими командами (или командами, перечисленными ниже).

Когда клиент подключён к базе данных, он может воспользоваться двумя дополнительными командами SQL (и равнозначными функциями), которые предоставляет Postgres Pro для управления параметрами конфигурации:

- Команда SHOW позволяет узнать текущее значение всех параметров. Соответствующая ей функция — `current_setting(имя_параметра text)`.
- Команда SET позволяет изменить текущее значение параметров, которые действуют локально в рамках сеанса; на другие сеансы она не влияет. Соответствующая ей функция — `set_config(имя_параметра, новое_значение, локально)`.

Кроме того, просмотреть и изменить значения параметров для текущего сеанса можно в системном представлении `pg_settings`:

- Запрос на чтение представления выдаёт ту же информацию, что и `SHOW ALL`, но более подробно. Этот подход и более гибкий, так как в нём можно указать условия фильтра или связать результат с другими отношениями.
- Выполнение UPDATE для этого представления, а именно присваивание значения столбцу `setting`, равносильно выполнению команды `SET`. Например, команде

  ```
  SET configuration_parameter TO DEFAULT;
  ```

  равнозначен запрос:

  ```
  UPDATE pg_settings SET setting = reset_val WHERE name = 'configuration_parameter';
  ```

18.1.4. Управление параметрами в командной строке

Помимо изменения глобальных значений по умолчанию и переопределения их на уровне базы данных или роли, параметры Postgres Pro можно изменить, используя средства командной строки. Управление через командную строку поддерживают и сервер, и клиентская библиотека libpq.

- При запуске сервера, значения параметров можно передать команде `postgres` в аргументе командной строки `-c`. Например:

  ```
  postgres -c log_connections=yes -c log_destination='syslog'
  ```

  Параметры, заданные таким образом, переопределяют те, что были установлены в `postgresql.conf` или командой `ALTER SYSTEM`, так что их нельзя изменить глобально без перезапуска сервера.
- При запуске клиентского сеанса, использующего libpq, значения параметров можно указать в переменной окружения `PGOPTIONS`. Заданные таким образом параметры будут определять значения по умолчанию на время сеанса, но никак не влияют на другие сеансы. По историческим причинам формат `PGOPTIONS` похож на тот, что применяется при запуске команды `postgres`; в частности, в нём должен присутствовать флаг `-c`. Например:

  ```
  env PGOPTIONS="-c geqo=off -c statement_timeout=5min" psql
  ```

  Другие клиенты и библиотеки могут иметь собственные механизмы управления параметрами, через командную строку или как-то иначе, используя которые пользователь сможет менять параметры сеанса, не выполняя непосредственно команды SQL.

18.1.5. Упорядочение содержимого файлов конфигурации

Postgres Pro предоставляет несколько возможностей для разделения сложных файлов `postgresql.conf` на вложенные файлы. Эти возможности особенно полезны при управлении множеством серверов с похожими, но не одинаковыми конфигурациями.

Помимо присваиваний значений параметров, `postgresql.conf` может содержать *директивы включения* файлов, которые будут прочитаны и обработаны, как если бы их содержимое было вставлено в данном месте файла конфигурации. Это позволяет разбивать файл конфигурации на физически отдельные части. Директивы включения записываются просто:

```
include 'имя_файла'
```

Если имя файла задаётся не абсолютным путём, оно рассматривается относительно каталога, в котором находится включающий файл конфигурации. Включения файлов могут быть вложенными.

Кроме того, есть директива `include_if_exists`, которая работает подобно `include`, за исключением случаев, когда включаемый файл не существует или не может быть прочитан. Обычная директива `include` считает это критической ошибкой, но `include_if_exists` просто выводит сообщение и продолжает обрабатывать текущий файл конфигурации.

Файл `postgresql.conf` может также содержать директивы `include_dir`, позволяющие подключать целые каталоги с файлами конфигурации. Они записываются так:

```
include_dir 'каталог'
```

Имена, заданные не абсолютным путём, рассматриваются относительно каталога, содержащего текущий файл конфигурации. В заданном каталоге включению подлежат только файлы с именами, оканчивающимися на `.conf`. При этом файлы с именами, начинающимися с «.», тоже игнорируются, для предотвращения ошибок, так как они считаются скрытыми в ряде систем. Набор файлов во включаемом каталоге обрабатывается по порядку имён (определяемому правилами, принятыми в C, т. е. цифры идут перед буквами, а буквы в верхнем регистре — перед буквами в нижнем).

Включение файлов или каталогов позволяет разделить конфигурацию базы данных на логические части, а не вести один большой файл `postgresql.conf`. Например, представьте, что в некоторой компании есть два сервера баз данных, с разным объёмом ОЗУ. Скорее всего при этом их конфигурации будут иметь общие элементы, например, параметры ведения журналов. Но параметры, связанные с памятью, у них будут различаться. Кроме того, другие параметры могут быть специфическими для каждого сервера. Один из вариантов эффективного управления такими конфигурациями — разделить изменения стандартной конфигурации на три файла. Чтобы подключить эти файлы, можно добавить в конец файла `postgresql.conf` следующие директивы:

```
include 'shared.conf'
include 'memory.conf'
include 'server.conf'
```

Общие для всех серверов параметры будут помещаться в `shared.conf`. Файл `memory.conf` может иметь два варианта — первый для серверов с 8ГБ ОЗУ, а второй для серверов с 16 ГБ. Наконец, `server.conf` может содержать действительно специфические параметры для каждого отдельного сервера.

Также возможно создать каталог с файлами конфигурации и поместить туда все эти файлы. Например, так можно подключить каталог `conf.d` в конце `postgresql.conf`:

```
include_dir 'conf.d'
```

Затем можно дать файлам в каталоге `conf.d` следующие имена:

```
00shared.conf
01memory.conf
02server.conf
```

Такое именование устанавливает чёткий порядок подключения этих файлов, что важно, так как если параметр определяется несколько раз в разных файлах конфигурации, действовать будет последнее определение. В рамках данного примера, установленное в `conf.d/02server.conf` значение переопределит значение того же параметра, заданное в `conf.d/01memory.conf`.

Вы можете применить этот подход и с описательными именами файлов:

```
00shared.conf
01memory-8GB.conf
02server-foo.conf
```

При таком упорядочивании каждому варианту файла конфигурации даётся уникальное имя. Это помогает исключить конфликты, если конфигурации разных серверов нужно хранить в одном месте, например, в репозитории системы управления версиями. (Кстати, хранение файлов конфигурации в системе управления версиями — это ещё один эффективный приём, который стоит применять.)

18.2. Расположения файлов

В дополнение к вышеупомянутому `postgresql.conf`, Postgres Pro обрабатывает два редактируемых вручную файла конфигурации, в которых настраивается аутентификация клиентов (их использование рассматривается в Главе 19). По умолчанию все три файла конфигурации размещаются в каталоге данных кластера БД. Параметры, описанные в этом разделе, позволяют разместить их и в любом другом месте. (Это позволяет упростить администрирование, в частности, выполнять резервное копирование этих файлов обычно проще, когда они хранятся отдельно.)

`data_directory` (`string`)

Задаёт каталог, в котором хранятся данные. Этот параметр можно задать только при запуске сервера.

`config_file` (`string`)

Задаёт основной файл конфигурации сервера (его стандартное имя — `postgresql.conf`). Этот параметр можно задать только в командной строке `postgres`.

`hba_file` (`string`)

Задаёт файл конфигурации для аутентификации по сетевым узлам (его стандартное имя — `pg_hba.conf`). Этот параметр можно задать только при старте сервера.

`ident_file` (`string`)

Задаёт файл конфигурации для сопоставлений имён пользователей (его стандартное имя — `pg_ident.conf`). Этот параметр можно задать только при запуске сервера. См. также Раздел 19.2.

`external_pid_file` (`string`)

Задаёт имя дополнительного файла с идентификатором процесса (PID), который будет создавать сервер для использования программами администрирования. Этот параметр можно задать только при запуске сервера.

При стандартной установке ни один из этих параметров не задаётся явно. Вместо них задаётся только каталог данных, аргументом командной строки `-D` или переменной окружения `PGDATA`, и все необходимые файлы конфигурации загружаются из этого каталога.

Если вы хотите разместить файлы конфигурации не в каталоге данных, то аргумент командной строки `postgres -D` или переменная окружения `PGDATA` должны указывать на каталог, содержащий файлы конфигурации, а в `postgresql.conf` (или в командной строке) должен задаваться параметр `data_directory`, указывающий, где фактически находятся данные. Учтите, что `data_directory` переопределяет путь, задаваемый в `-D` или `PGDATA` как путь каталога данных, но не расположение файлов конфигурации.

При желании вы можете задать имена и пути файлов конфигурации по отдельности, воспользовавшись параметрами `config_file`, `hba_file` и/или `ident_file`. Параметр `config_file` можно задать только в командной строке `postgres`, тогда как остальные можно задать и в основном файле конфигурации. Если явно заданы все три эти параметра плюс `data_directory`, то задавать `-D` или `PGDATA` не нужно.

Во всех этих параметрах относительный путь должен задаваться от каталога, в котором запускается `postgres`.

18.3. Подключения и аутентификация

18.3.1. Параметры подключений

`listen_addresses` (`string`)

Задаёт адреса TCP/IP, по которым сервер будет принимать подключения клиентских приложений. Это значение принимает форму списка, разделённого запятыми, из имён и/или числовых IP-адресов компьютеров. Особый элемент, `*`, обозначает все имеющиеся IP-интерфейсы. Запись `0.0.0.0` позволяет задействовать все адреса IPv4, а `::` — все адреса IPv6. Если список пуст, сервер не будет привязываться ни к какому IP-интерфейсу, а значит, подключиться к нему можно будет только через Unix-сокеты. По умолчанию этот параметр содержит localhost, что допускает подключение к серверу по TCP/IP только через локальный интерфейс «замыкания». Хотя механизм аутентификации клиентов (см. Главу 19) позволяет гибко управлять доступом пользователей к серверу, параметр `listen_addresses` может ограничить интерфейсы, через которые будут приниматься соединения, что бывает полезно для предотвращения злонамеренных попыток подключения через незащищённые сетевые интерфейсы. Этот параметр можно задать только при запуске сервера.

`port` (`integer`)

TCP-порт, открываемый сервером; по умолчанию, 5432. Заметьте, что этот порт используется для всех IP-адресов, через которые сервер принимает подключения. Этот параметр можно задать только при запуске сервера.

`max_connections` (`integer`)

Определяет максимальное число одновременных подключений к серверу БД. По умолчанию обычно это 100 подключений, но это число может быть меньше, если ядро накладывает свои ограничения (это определяется в процессе initdb). Этот параметр можно задать только при запуске сервера.

Для ведомого сервера значение этого параметра должно быть больше или равно значению на ведущем. В противном случае на ведомом сервере не будут разрешены запросы.

`superuser_reserved_connections` (`integer`)

Определяет количество «слотов» подключений, которые Postgres Pro будет резервировать для суперпользователей. При этом всего одновременно активными могут быть максимум max_connections подключений. Когда число активных одновременных подключений больше или равно `max_connections` минус `superuser_reserved_connections`, принимаются только подключения суперпользователей, а все другие подключения, в том числе подключения для репликации, запрещаются.

По умолчанию резервируются три соединения. Это значение должно быть меньше значения `max_connections`. Задать этот параметр можно только при запуске сервера.

`unix_socket_directories` (`string`)

Задаёт каталог Unix-сокета, через который сервер будет принимать подключения клиентских приложений. Создать несколько сокетов можно, перечислив в этом значении несколько каталогов через запятую. Пробелы между элементами этого списка игнорируются; если в пути каталога содержатся пробелы, его нужно заключать в двойные кавычки. При пустом значении сервер не будет работать с Unix-сокетами, в этом случае к нему можно будет подключиться только по TCP/IP. Значение по умолчанию обычно `/tmp`, но его можно изменить во время сборки. Задать этот параметр можно только при запуске сервера.

Помимо самого файла сокета, который называется `.s.PGSQL.`*`nnnn`* (где *`nnnn`* — номер порта сервера), в каждом каталоге `unix_socket_directories` создаётся обычный файл `.s.PGSQL.`*`nnnn`*`.lock`. Ни в коем случае не удаляйте эти файлы вручную.

Этот параметр не действует в системе Windows, так как в ней нет Unix-сокетов.

`unix_socket_group` (`string`)

Задаёт группу-владельца Unix-сокетов. (Пользователем-владельцем сокетов всегда будет пользователь, запускающий сервер.) В сочетании с `unix_socket_permissions` данный параметр можно использовать как дополнительный механизм управления доступом к Unix-сокетам. По умолчанию он содержит пустую строку, то есть группой-владельцем становится основная группа пользователя, запускающего сервер. Задать этот параметр можно только при запуске сервера.

Этот параметр не действует в системе Windows, так как в ней нет Unix-сокетов.

`unix_socket_permissions` (`integer`)

Задаёт права доступа к Unix-сокетам. Для Unix-сокетов применяется обычный набор разрешений Unix. Значение параметра ожидается в числовом виде, который принимают функции `chmod` и `umask`. (Для применения обычного восьмеричного формата число должно начинаться с `0` (нуля).)

По умолчанию действуют разрешения `0777`, при которых подключаться к сокету могут все. Другие разумные варианты — `0770` (доступ имеет только пользователь и группа, см. также `unix_socket_group`) и `0700` (только пользователь). (Заметьте, что для Unix-сокетов требуется только право на запись, так что добавлять или отзывать права на чтение/выполнение не имеет смысла.)

Этот механизм управления доступом не зависит от описанного в Главе 19.

Этот параметр можно задать только при запуске сервера.

Данный параметр неприменим для некоторых систем, в частности, Solaris (а именно Solaris 10), которые полностью игнорируют разрешения для сокетов. В таких системах примерно тот же эффект можно получить, указав в параметре `unix_socket_directories` каталог, доступ к которому ограничен должным образом. Этот параметр также неприменим в Windows, где нет Unix-сокетов.

`bonjour` (`boolean`)

Включает объявления о существовании сервера посредством Bonjour. По умолчанию выключен. Задать этот параметр можно только при запуске сервера.

`bonjour_name` (`string`)

Задаёт имя службы в среде Bonjour. Если значение этого параметра — пустая строка (`''`) (это значение по умолчанию), в качестве этого имени используется имя компьютера. Этот параметр игнорируется, если сервер был скомпилирован без поддержки Bonjour. Задать этот параметр можно только при запуске сервера.

`tcp_keepalives_idle` (`integer`)

Задаёт период неактивности (в секундах), после которого по TCP клиенту должен отправляться сигнал сохранения соединения. При значении 0 действует системный параметр. Этот параметр поддерживается только в системах, воспринимающих параметр сокета `TCP_KEEPIDLE` или равнозначный, а также в Windows; в других системах он должен быть равен нулю. В сеансах, подключённых через Unix-сокеты, он игнорируется и всегда считается равным 0.

> **Примечание**
>
> В Windows при нулевом значении этот период устанавливается равным 2 часам, так как Windows не позволяет прочитать системное значение по умолчанию.

`tcp_keepalives_interval` (`integer`)

Задаёт интервал (в секундах), по истечении которого следует повторять сигнал сохранения соединения, если ответ от клиента не был получен. При значении 0 действует системная величина. Этот параметр поддерживается только в системах, воспринимающих параметр сокета `TCP_KEEPINTVL` или равнозначный, а также в Windows; в других системах он должен быть равен нулю. В сеансах, подключённых через Unix-сокеты, он игнорируется и всегда считается равным 0.

> **Примечание**
>
> В Windows при нулевом значении этот интервал устанавливается равным 1 секунде, так как Windows не позволяет прочитать системное значение по умолчанию.

`tcp_keepalives_count` (`integer`)

Задаёт число TCP-сигналов сохранения соединения, которые могут быть потеряны до того, как соединение сервера с клиентом будет признано прерванным. При значении 0 действует системная величина. Этот параметр поддерживается только в системах, воспринимающих параметр сокета `TCP_KEEPCNT` или равнозначный; в других системах он должен быть равен нулю. В сеансах, подключённых через Unix-сокеты, он игнорируется и всегда считается равным 0.

> **Примечание**
>
> В Windows данный параметр не поддерживается и должен быть равен нулю.

18.3.2. Безопасность и аутентификация

`authentication_timeout` (`integer`)

Максимальное время, за которое должна произойти аутентификация (в секундах). Если потенциальный клиент не сможет пройти проверку подлинности за это время, сервер закроет соединение. Благодаря этому зависшие при подключении клиенты не будут занимать соединения неограниченно долго. Значение этого параметра по умолчанию — одна минута (`1m`). Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`ssl` (`boolean`)

Разрешает подключения SSL. Прежде чем его использовать, прочитайте Раздел 17.9. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. Значение по умолчанию — `off`.

`ssl_ca_file` (`string`)

Задаёт имя файла, содержащего сертификаты центров сертификации (ЦС) для SSL-сервера. При указании относительного пути он рассматривается от каталога данных. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. По умолчанию этот параметр пуст, что означает, что файл сертификатов ЦС не загружается и проверка клиентских сертификатов не производится.

В предыдущих выпусках Postgres Pro это имя было неизменяемым — `root.crt`.

`ssl_cert_file` (`string`)

Задаёт имя файла, содержащего сертификат этого SSL-сервера. При указании относительного пути он рассматривается от каталога данных. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. Значение по умолчанию — `server.crt`.

`ssl_crl_file` (`string`)

Задаёт имя файла, содержащего список отзыва клиентских сертификатов (CRL, Certificate Revocation List) для SSL. При указании относительного пути он рассматривается от каталога данных. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. По умолчанию этот параметр пуст, что означает, что файл CRL не загружается.

В предыдущих выпусках Postgres Pro имя этого файла было неизменяемым — `root.crl`.

`ssl_key_file` (`string`)

Задаёт имя файла, содержащего закрытый ключ SSL-сервера. При указании относительного пути он рассматривается от каталога данных. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. Значение по умолчанию — `server.key`.

`ssl_ciphers` (`string`)

Задаёт список наборов шифров SSL, которые могут применяться для SSL-соединений. Синтаксис этого параметра и список поддерживаемых значений можно найти на странице ciphers руководства по OpenSSL. Этот параметр действует только для подключений TLS версии 1.2 и ниже. Для подключений TLS версии 1.3 возможность выбора шифров в настоящее время отсутствует. Значение по умолчанию — `HIGH:MEDIUM:+3DES:!aNULL`. Обычно оно вполне приемлемо при отсутствии особых требований по безопасности.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

Объяснение значения по умолчанию:

`HIGH`

Наборы шифров, в которых используются шифры из группы высокого уровня (`HIGH`), (например: AES, Camellia, 3DES)

`MEDIUM`

Наборы шифров, в которых используются шифры из группы среднего уровня (`MEDIUM`) (например, RC4, SEED)

`+3DES`

Порядок шифров для группы `HIGH` по умолчанию в OpenSSL определён некорректно. В нём 3DES оказывается выше AES128, что неправильно, так как он считается менее безопасным, чем AES128, и работает гораздо медленнее. Включение `+3DES` меняет этот порядок, чтобы данный алгоритм следовал после всех шифров групп `HIGH` и `MEDIUM`.

`!aNULL`

Отключает наборы анонимных шифров, не требующие проверки подлинности. Такие наборы уязвимы для атак с посредником, поэтому использовать их не следует.

Конкретные наборы шифров и их свойства очень различаются от версии к версии OpenSSL. Чтобы получить фактическую информацию о них для текущей установленной версии OpenSSL, выполните команду `openssl ciphers -v 'HIGH:MEDIUM:+3DES:!aNULL'`. Учтите, что этот список фильтруется во время выполнения, в зависимости от типа ключа сервера.

`ssl_prefer_server_ciphers` (`boolean`)

Определяет, должны ли шифры SSL сервера предпочитаться клиентским. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. Значение по умолчанию — `true`.

В старых версиях Postgres Pro этот параметр отсутствовал и предпочтение отдавалось выбору клиента. Введён этот параметр в основном для обеспечения совместимости с этими версиями. Вообще же обычно лучше использовать конфигурацию сервера, так как в конфигурации на стороне клиента более вероятны ошибки.

`ssl_ecdh_curve` (`string`)

Задаёт имя кривой для использования при обмене ключами ECDH. Эту кривую должны поддерживать все подключающиеся клиенты. Это не обязательно должна быть кривая, с которой был получен ключ сервера. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. Значение по умолчанию — `prime256v1`.

Наиболее распространённые кривые в OpenSSL — `prime256v1` (NIST P-256), `secp384r1` (NIST P-384), `secp521r1` (NIST P-521). Полный список доступных кривых можно получить командой `openssl ecparam -list_curves`. Однако не все из них пригодны для TLS.

`password_encryption` (`enum`)

Когда в CREATE ROLE или ALTER ROLE задаётся пароль, этот параметр определяет, каким алгоритмом его шифровать. Значение по умолчанию — `md5` (пароль сохраняется в виде хеша MD5), также в качестве псевдонима `md5` принимается значение `on`. Значение `scram-sha-256` указывает, что пароль будет шифроваться алгоритмом SCRAM-SHA-256.

Учтите, что старые клиенты могут не поддерживать механизм проверки подлинности SCRAM и поэтому не будут работать с паролями, зашифрованными алгоритмом SCRAM-SHA-256. За подробностями обратитесь к Подразделу 19.3.2.

`ssl_dh_params_file` (`string`)

Задаёт имя файла с параметрами алгоритма Диффи-Хеллмана, применяемого для так называемого эфемерного семейства DH шифров SSL. По умолчанию значение пустое, то есть используются стандартные параметры DH, заданные при компиляции. Использование нестандартных параметров DH защищает от атаки, рассчитанной на взлом хорошо известных встроенных параметров DH. Создать собственный файл с параметрами DH можно, выполнив команду `openssl dhparam -out dhparams.pem 2048`.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`krb_server_keyfile` (`string`)

Задаёт размещение файла ключей для сервера Kerberos. За подробностями обратитесь к Подразделу 19.3.3. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`krb_caseins_users` (`boolean`)

Определяет, должны ли имена пользователей GSSAPI обрабатываться без учёта регистра. По умолчанию значение этого параметра — `off` (регистр учитывается). Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`db_user_namespace` (`boolean`)

Этот параметр позволяет относить имена пользователей к базам данных. По умолчанию он имеет значение off (выключен). Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

Если он включён, имена создаваемых пользователей должны иметь вид *`имя_пользователя@база_данных`*. Когда подключающийся клиент передаёт *`имя_пользователя`*, к этому имени добавляется `@` с именем базы данных, и сервер идентифицирует пользователя по этому полному имени. Заметьте, что для создания пользователя с именем, содержащим `@`, в среде SQL потребуется заключить это имя в кавычки.

Когда этот параметр включён, он не мешает создавать и использовать обычных глобальных пользователей. Чтобы подключиться с таким именем пользователя, достаточно добавить к имени `@`, например так: `joe@`. Получив такое имя, сервер отбросит `@`, и будет идентифицировать пользователя по начальному имени.

Параметр `db_user_namespace` порождает расхождение между именами пользователей на стороне сервера и клиента. Но проверки подлинности всегда выполняются с именем с точки зрения сервера, так что, настраивая аутентификацию, следует указывать серверное представление имени, а не клиентское. Так как метод аутентификации `md5` подмешивает имя пользователя в качестве соли и на стороне сервера, и на стороне клиента, при включённом параметре `db_user_namespace` использовать `md5` невозможно.

> **Примечание**
>
> Эта возможность предлагается в качестве временной меры, пока не будет найдено полноценное решение. Тогда этот параметр будет ликвидирован.

18.4. Потребление ресурсов

18.4.1. Память

`shared_buffers` (`integer`)

Задаёт объём памяти, который будет использовать сервер баз данных для буферов в разделяемой памяти. По умолчанию это обычно 128 мегабайт (`128MB`), но может быть и меньше, если конфигурация вашего ядра накладывает дополнительные ограничения (это определяется в процессе initdb). Это значение не должно быть меньше 128 килобайт. (Этот минимум зависит от величины `BLCKSZ`.) Однако для хорошей производительности обычно требуются гораздо большие значения. Задать этот параметр можно только при запуске сервера.

Если вы используете выделенный сервер с объёмом ОЗУ 1 ГБ и более, разумным начальным значением `shared_buffers` будет 25% от объёма памяти. Существуют варианты нагрузки, при которых эффективны будут и ещё большие значения `shared_buffers`, но так как Postgres Pro использует и кеш операционной системы, выделять для `shared_buffers` более 40% ОЗУ вряд ли будет полезно. При увеличении `shared_buffers` обычно требуется соответственно увеличить `max_wal_size`, чтобы растянуть процесс записи большого объёма новых или изменённых данных на более продолжительное время.

В системах с объёмом ОЗУ меньше 1 ГБ стоит ограничиться меньшим процентом ОЗУ, чтобы оставить достаточно места операционной системе.

`huge_pages` (`enum`)

Включает/отключает использование огромных страниц памяти. Допустимые значения: `try` (попытаться, по умолчанию), `on` (вкл.) и `off` (выкл.).

В настоящее время это поддерживается только в Linux. В других системах значение `try` просто игнорируется.

В результате использования огромных страниц уменьшаются таблицы страниц и сокращается время, которое тратит процессор на управление памятью. За подробностями обратитесь к Подразделу 17.4.5.

Когда `huge_pages` имеет значение `try`, сервер попытается использовать огромные страницы, но может переключиться на обычные, если это не удастся. Со значением `on`, если использовать огромные страницы не получится, сервер не будет запущен. Со значением `off` огромные страницы использоваться не будут.

temp_buffers (integer)

Задаёт максимальное число временных буферов для каждого сеанса, По умолчанию объём временных буферов составляет восемь мегабайт (1024 буфера). Этот параметр можно изменить в отдельном сеансе, но только до первого обращения к временным таблицам; после этого изменить его значение для текущего сеанса не удастся.

Сеанс выделяет временные буферы по мере необходимости до достижения предела, заданного параметром temp_buffers. Если сеанс не задействует временные буферы, то для него хранятся только дескрипторы буферов, которые занимают около 64 байт (в количестве temp_buffers). Однако если буфер действительно используется, он будет дополнительно занимать 8192 байта (или BLCKSZ байт, в общем случае).

max_prepared_transactions (integer)

Задаёт максимальное число транзакций, которые могут одновременно находиться в «подготовленном» состоянии (см. PREPARE TRANSACTION). При нулевом значении (по умолчанию) механизм подготовленных транзакций отключается. Задать этот параметр можно только при запуске сервера.

Если использовать транзакции не планируется, этот параметр следует обнулить, чтобы не допустить непреднамеренного создания подготовленных транзакций. Если же подготовленные транзакции применяются, то max_prepared_transactions, вероятно, должен быть не меньше, чем max_connections, чтобы подготовить транзакцию можно было в каждом сеансе.

Для ведомого сервера значение этого параметра должно быть больше или равно значению на ведущем. В противном случае на ведомом сервере не будут разрешены запросы.

work_mem (integer)

Задаёт объём памяти, который будет использоваться для внутренних операций сортировки и хеш-таблиц, прежде чем будут задействованы временные файлы на диске. Значение по умолчанию — четыре мегабайта (4MB). Заметьте, что в сложных запросах одновременно могут выполняться несколько операций сортировки или хеширования, и при этом указанный объём памяти может использоваться в каждой операции, прежде чем данные начнут вытесняться во временные файлы. Кроме того, такие операции могут выполняться одновременно в разных сеансах. Таким образом, общий объём памяти может многократно превосходить значение work_mem; это следует учитывать, выбирая подходящее значение. Операции сортировки используются для ORDER BY, DISTINCT и соединений слиянием. Хеш-таблицы используются при соединениях и агрегировании по хешу, а также обработке подзапросов IN с применением хеша.

maintenance_work_mem (integer)

Задаёт максимальный объём памяти для операций обслуживания БД, в частности VACUUM, CREATE INDEX и ALTER TABLE ADD FOREIGN KEY. По умолчанию его значение — 64 мегабайта (64MB). Так как в один момент времени в сеансе может выполняться только одна такая операция и обычно они не запускаются параллельно, это значение вполне может быть гораздо больше work_mem. Увеличение этого значения может привести к ускорению операций очистки и восстановления БД из копии.

Учтите, что когда выполняется автоочистка, этот объём может быть выделен autovacuum_max_workers раз, поэтому не стоит устанавливать значение по умолчанию слишком большим. Возможно, будет лучше управлять объёмом памяти для автоочистки отдельно, изменяя autovacuum_work_mem.

Заметьте, что для сбора идентификаторов мёртвых кортежей VACUUM может использовать не более 1GB памяти.

replacement_sort_tuples (integer)

Когда количество сортируемых кортежей меньше заданного числа, для получения первого потока сортируемых данных будет применяться не алгоритм quicksort, а выбор с замещением.

Это может быть полезно в системах с ограниченным объёмом памяти, когда кортежи, поступающие на вход большой операции сортировки, характеризуются хорошей корреляцией физического и логического порядка. Заметьте, что это не относится к кортежам с *обратной* корреляцией. Вполне возможно, что алгоритм выбора с замещением сформирует один длинный поток и слияние не потребуется, тогда как со стратегией по умолчанию может быть получено много потоков, которые затем потребуется слить для получения окончательного отсортированного результата. Это позволяет ускорить операции сортировки.

Значение по умолчанию — 150000 кортежей. Заметьте, что увеличивать это значение обычно не очень полезно, и может быть даже контрпродуктивно, так как эффективность приоритетной очереди зависит от доступного объёма кеша процессора, тогда как со стандартной стратегией потоки сортируются *кеш-независимым* алгоритмом. Благодаря этому стандартная стратегия позволяет автоматически и прозрачно использовать доступный кеш процессора более эффективным образом.

Если в `maintenance_work_mem` задано значение по умолчанию, внешние сортировки в служебных командах (например, сортировки, выполняемые командами `CREATE INDEX` для построения-индекса B-дерева) обычно никогда не используют алгоритм выбора с замещением (так как все кортежи помещаются в память), кроме случаев, когда входные кортежи достаточно велики.

`autovacuum_work_mem` (`integer`)

Задаёт максимальный объём памяти, который будет использовать каждый рабочий процесс автоочистки. При действующем по умолчанию значении `-1` этот объём определяется значением maintenance_work_mem. Этот параметр не влияет на поведение команды `VACUUM`, выполняемой в других контекстах. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

Для сбора идентификаторов мёртвых кортежей автоочистка может использовать максимум `1GB` памяти, поэтому увеличение `autovacuum_work_mem` до большего значения не повлияет на количество мёртвых кортежей, которые автоочистка может собрать при сканировании таблицы.

`max_stack_depth` (`integer`)

Задаёт максимальную безопасную глубину стека для исполнителя. В идеале это значение должно равняться предельному размеру стека, ограниченному ядром (который устанавливается командой `ulimit -s` или аналогичной), за вычетом запаса примерно в мегабайт. Этот запас необходим, потому что сервер проверяет глубину стека не в каждой процедуре, а только в потенциально рекурсивных процедурах, например, при вычислении выражений. Значение по умолчанию — два мегабайта (`2MB`) — выбрано с большим запасом, так что риск переполнения стека минимален. Но с другой стороны, его может быть недостаточно для выполнения сложных функций. Изменить этот параметр могут только суперпользователи.

Если `max_stack_depth` будет превышать фактический предел ядра, то функция с неограниченной рекурсией сможет вызвать крах отдельного процесса сервера. В системах, где Postgres Pro может определить предел, установленный ядром, он не позволит установить для этого параметра небезопасное значение. Однако эту информацию выдают не все системы, поэтому выбирать это значение следует с осторожностью.

`dynamic_shared_memory_type` (`enum`)

Выбирает механизм динамической разделяемой памяти, который будет использоваться сервером. Допустимые варианты: `posix` (для выделения разделяемой памяти POSIX функцией `shm_open`), `sysv` (для выделения разделяемой памяти System V функцией `shmget`), `windows` (для выделения разделяемой памяти в Windows), `mmap` (для эмуляции разделяемой памяти через отображение в память файлов, хранящихся в каталоге данных) и `none` (для отключения этой функциональности). Не все варианты поддерживаются на разных платформах; первый из поддерживаемых данной платформой вариантов становится для неё вариантом по умолчанию. Применять `mmap`, который нигде не выбирается по умолчанию, вообще не рекомендуется, так

как операционная система может периодически записывать на диск изменённые страницы, что создаст дополнительную нагрузку; однако это может быть полезно для отладки, когда каталог pg_dynshmem находится на RAM-диске или когда другие механизмы разделяемой памяти недоступны.

18.4.2. Диск

temp_file_limit (integer)

Задаёт максимальный объём дискового пространства, который сможет использовать один процесс для временных файлов, например, при сортировке и хешировании, или для сохранения удерживаемого курсора. Транзакция, которая попытается превысить этот предел, будет отменена. Этот параметр задаётся в килобайтах, а значение -1 (по умолчанию) означает, что предел отсутствует. Изменить этот параметр могут только суперпользователи.

Этот параметр ограничивает общий объём, который могут занимать в момент времени все временные файлы, задействованные в данном процессе Postgres Pro. Следует отметить, что это *не* касается файлов явно создаваемых временных таблиц; ограничивается только объём временных файлов, которые создаются неявно при выполнении запросов.

18.4.3. Использование ресурсов ядра

max_files_per_process (integer)

Задаёт максимальное число файлов, которые могут быть одновременно открыты каждым серверным подпроцессом. Значение по умолчанию — 1000 файлов. Если ядро реализует безопасное ограничение по процессам, об этом параметре можно не беспокоиться. Но на некоторых платформах (а именно, в большинстве систем BSD) ядро позволяет отдельному процессу открыть больше файлов, чем могут открыть несколько процессов одновременно. Если вы столкнётесь с ошибками «Too many open files» (Слишком много открытых файлов), попробуйте уменьшить это число. Задать этот параметр можно только при запуске сервера.

18.4.4. Задержка очистки по стоимости

Во время выполнения команд VACUUM и ANALYZE система ведёт внутренний счётчик, в котором суммирует оцениваемую стоимость различных выполняемых операций ввода/вывода. Когда накопленная стоимость превышает предел (vacuum_cost_limit), процесс, выполняющий эту операцию, засыпает на некоторое время (vacuum_cost_delay). Затем счётчик сбрасывается и процесс продолжается.

Данный подход реализован для того, чтобы администраторы могли снизить влияние этих команд на параллельную работу с базой, за счёт уменьшения нагрузки на подсистему ввода-вывода. Очень часто не имеет значения, насколько быстро выполнятся команды обслуживания (например, VACUUM и ANALYZE), но очень важно, чтобы они как можно меньше влияли на выполнение других операций с базой данных. Администраторы имеют возможность управлять этим, настраивая задержку очистки по стоимости.

По умолчанию этот режим отключён для выполняемых вручную команд VACUUM. Чтобы включить его, нужно установить в vacuum_cost_delay ненулевое значение.

vacuum_cost_delay (integer)

Продолжительность времени, в миллисекундах, в течение которого будет простаивать процесс, превысивший предел стоимости. По умолчанию его значение равно нулю, то есть задержка очистки отсутствует. При положительных значениях интенсивность очистки будет зависеть от стоимости. Заметьте, что во многих системах разрешение таймера составляет 10 мс, поэтому если задать в vacuum_cost_delay значение, не кратное 10, фактически будет получен тот же результат, что и со следующим за ним кратным 10.

При настройке интенсивности очистки для vacuum_cost_delay обычно выбираются довольно небольшие значения, например 10 или 20 миллисекунд. Чтобы точнее ограничить потребление ресурсов при очистке, лучше всего изменять другие параметры стоимости очистки.

vacuum_cost_page_hit (integer)

Примерная стоимость очистки буфера, оказавшегося в общем кеше. Это подразумевает блокировку пула буферов, поиск в хеш-таблице и сканирование содержимого страницы. По умолчанию этот параметр равен одному.

vacuum_cost_page_miss (integer)

Примерная стоимость очистки буфера, который нужно прочитать с диска. Это подразумевает блокировку пула буферов, поиск в хеш-таблице, чтение требуемого блока с диска и сканирование его содержимого. По умолчанию этот параметр равен 10.

vacuum_cost_page_dirty (integer)

Примерная стоимость очистки, при которой изменяется блок, не модифицированный ранее. В неё включается дополнительная стоимость ввода/вывода, связанная с записью изменённого блока на диск. По умолчанию этот параметр равен 20.

vacuum_cost_limit (integer)

Общая стоимость, при накоплении которой процесс очистки будет засыпать. По умолчанию этот параметр равен 200.

> **Примечание**
>
> Некоторые операции устанавливают критические блокировки и поэтому должны завершаться как можно быстрее. Во время таких операций задержка очистки по стоимости не осуществляется, так что накопленная за это время стоимость может намного превышать установленный предел. Во избежание ненужных длительных задержек в таких случаях, фактическая задержка вычисляется по формуле vacuum_cost_delay * accumulated_balance / vacuum_cost_limit и ограничивается максимумом, равным vacuum_cost_delay * 4.

18.4.5. Фоновая запись

В числе специальных процессов сервера есть процесс *фоновой записи*, задача которого — осуществлять запись «грязных» (новых или изменённых) общих буферов на диск. Когда количество чистых общих буферов считается недостаточным, данный процесс записывает грязные буферы в файловую систему и помечает их как чистые. Это снижает вероятность того, что серверные процессы, выполняющие запросы пользователей, не смогут найти чистые буферы и им придётся сбрасывать грязные буферы самостоятельно. Однако процесс фоновой записи увеличивает общую нагрузку на подсистему ввода/вывода, так как он может записывать неоднократно изменяемую страницу несколько раз, тогда как её можно было бы записать всего один раз в контрольной точке. Параметры, рассматриваемые в данном подразделе, позволяют настроить поведение фоновой записи для конкретных нужд.

bgwriter_delay (integer)

Задаёт задержку между раундами активности процесса фоновой записи. Во время раунда этот процесс осуществляет запись некоторого количества загрязнённых буферов (это настраивается следующими параметрами). Затем он засыпает на время bgwriter_delay (задаваемое в миллисекундах), и всё повторяется снова. Однако если в пуле не остаётся загрязнённых буферов, он может быть неактивен более длительное время. По умолчанию этот параметр равен 200 миллисекундам (200ms). Заметьте, что во многих системах разрешение таймера составляет 10 мс, поэтому если задать в bgwriter_delay значение, не кратное 10, фактически будет получен тот же результат, что и со следующим за ним кратным 10. Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера.

bgwriter_lru_maxpages (integer)

Задаёт максимальное число буферов, которое сможет записать процесс фоновой записи за раунд активности. При нулевом значении фоновая запись отключается. (Учтите, что на

контрольные точки, которые управляются отдельным вспомогательным процессом, это не влияет.) По умолчанию значение этого параметра — 100 буферов. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`bgwriter_lru_multiplier` (`floating point`)

Число загрязнённых буферов, записываемых в очередном раунде, зависит от того, сколько новых буферов требовалось серверным процессам в предыдущих раундах. Средняя недавняя потребность умножается на `bgwriter_lru_multiplier` и предполагается, что именно столько буферов потребуется на следующем раунде. Процесс фоновой записи будет записывать на диск и освобождать буферы, пока число свободных буферов не достигнет целевого значения. (При этом число буферов, записываемых за раунд, ограничивается сверху параметром `bgwriter_lru_maxpages`.) Таким образом, со множителем, равным 1.0, записывается ровно столько буферов, сколько требуется по предположению («точно по плану»). Увеличение этого множителя даёт некоторую страховку от резких скачков потребностей, тогда как уменьшение отражает намерение оставить некоторый объём записи для серверных процессов. По умолчанию он равен 2.0. Этот параметр можно установить только в файле `postgresql.conf` или в командной строке при запуске сервера.

`bgwriter_flush_after` (`integer`)

Когда процессом фоновой записи записывается больше чем `bgwriter_flush_after` байт, сервер даёт указание ОС произвести запись этих данных в нижележащее хранилище. Это ограничивает объём «грязных» данных в страничном кеше ядра и уменьшает вероятность затормаживания при выполнении `fsync` в конце контрольной точки или когда ОС сбрасывает данные на диск большими порциями в фоне. Часто это значительно уменьшает задержки транзакций, но бывают ситуации (особенно когда объём рабочей нагрузки больше shared_buffers, но меньше страничного кеша ОС), когда производительность может упасть. Этот параметр действует не на всех платформах. Он может принимать значение от `0` (при этом управление отложенной записью отключается) до 2 мегабайт (`2MB`). Значение по умолчанию — `512kB` в Linux и `0` в других ОС. (Если `BLCKSZ` отличен от 8 КБ, значение по умолчанию и максимум корректируются пропорционально.) Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

С маленькими значениями `bgwriter_lru_maxpages` и `bgwriter_lru_multiplier` уменьшается активность ввода/вывода со стороны процесса фоновой записи, но увеличивается вероятность того, что запись придётся производить непосредственно серверным процессам, что замедлит выполнение запросов.

18.4.6. Асинхронное поведение

`effective_io_concurrency` (`integer`)

Задаёт допустимое число параллельных операций ввода/вывода, которое говорит Postgres Pro о том, сколько операций ввода/вывода могут быть выполнены одновременно. Чем больше это число, тем больше операций ввода/вывода будет пытаться выполнить параллельно Postgres Pro в отдельном сеансе. Допустимые значения лежат в интервале от 1 до 1000, а нулевое значение отключает асинхронные запросы ввода/вывода. В настоящее время этот параметр влияет только на сканирование по битовой карте.

Для магнитных носителей хорошим начальным значением этого параметра будет число отдельных дисков, составляющих массив RAID 0 или RAID 1, в котором размещена база данных. (Для RAID 5 следует исключить один диск (как диск с чётностью).) Однако если база данных часто обрабатывает множество запросов в различных сеансах, и при небольших значениях дисковый массив может быть полностью загружен. Если продолжать увеличивать это значение при полной загрузке дисков, это приведёт только к увеличению нагрузки на процессор. Диски SSD и другие виды хранилища в памяти часто могут обрабатывать множество параллельных запросов, так что оптимальным числом может быть несколько сотен.

Асинхронный ввод/вывод зависит от эффективности функции `posix_fadvise`, которая отсутствует в некоторых операционных системах. В случае её отсутствия попытка задать

для этого параметра любое ненулевое значение приведёт к ошибке. В некоторых системах (например, в Solaris), эта функция присутствует, но на самом деле ничего не делает.

Значение по умолчанию равно 1 в системах, где это поддерживается, и 0 в остальных. Это значение можно переопределить для таблиц в определённом табличном пространстве, установив одноимённый параметр табличного пространства (см. ALTER TABLESPACE).

`max_worker_processes` (`integer`)

Задаёт максимальное число фоновых процессов, которое можно запустить в текущей системе. Этот параметр можно задать только при запуске сервера. Значение по умолчанию — 8.

Для ведомого сервера значение этого параметра должно быть больше или равно значению на ведущем. В противном случае на ведомом сервере не будут разрешены запросы.

При изменении этого значения также может быть полезно изменить max_parallel_workers и max_parallel_workers_per_gather.

`max_parallel_workers_per_gather` (`integer`)

Задаёт максимальное число рабочих процессов, которые могут запускаться одним узлом `Gather` или `Gather Merge`. Параллельные рабочие процессы берутся из пула процессов, контролируемого параметром max_worker_processes, в количестве, ограничиваемом значением max_parallel_workers. Учтите, что запрошенное количество рабочих процессов может быть недоступно во время выполнения. В этом случае план будет выполняться с меньшим числом процессов, что может быть неэффективно. Значение по умолчанию — 2. Значение 0 отключает параллельное выполнение запросов.

Учтите, что параллельные запросы могут потреблять значительно больше ресурсов, чем не параллельные, так как каждый рабочий процесс является отдельным процессом и оказывает на систему примерно такое же влияние, как дополнительный пользовательский сеанс. Это следует учитывать, выбирая значение этого параметра, а также настраивая другие параметры, управляющие использованием ресурсов, например work_mem. Ограничения ресурсов, такие как `work_mem`, применяются к каждому рабочему процессу отдельно, что означает, что общая нагрузка для всех процессов может оказаться гораздо больше, чем при обычном использовании одного процесса. Например, параллельный запрос, задействующий 4 рабочих процесса, может использовать в 5 раз больше времени процессора, объёма памяти, ввода/вывода и т. д., по сравнению с запросом, не задействующим рабочие процессы вовсе.

За дополнительными сведениями о параллельных запросах обратитесь к Главе 15.

`max_parallel_workers` (`integer`)

Задаёт максимальное число рабочих процессов, которое система сможет поддерживать для параллельных запросов. Значение по умолчанию — 8. При увеличении или уменьшения этого значения также может иметь смысл скорректировать max_parallel_workers_per_gather. Заметьте, что значение данного параметра, превышающее max_worker_processes, не будет действовать, так как параллельные рабочие процессы берутся из пула рабочих процессов, ограничиваемого этим параметром.

`backend_flush_after` (`integer`)

Когда одним обслуживающим процессом записывается больше `backend_flush_after` байт, сервер даёт указание ОС произвести запись этих данных в нижележащее хранилище. Это ограничивает объём «грязных» данных в страничном кеше ядра и уменьшает вероятность затормаживания при выполнении `fsync` в конце контрольной точки или когда ОС сбрасывает данные на диск большими порциями в фоне. Часто это значительно сокращает задержки транзакций, но бывают ситуации (особенно когда объём рабочей нагрузки больше shared_buffers, но меньше страничного кеша ОС), когда производительность может упасть. Этот параметр действует не на всех платформах. Он может принимать значение от `0` (при этом управление отложенной записью отключается) до 2 мегабайт (`2MB`). По умолчанию он имеет

значение 0, то есть это поведение отключено. (Если BLCKSZ отличен от 8 КБ, максимальное значение корректируется пропорционально.)

old_snapshot_threshold (integer)

Задаёт минимальное время, которое можно пользоваться снимком без риска получить ошибку снимок слишком стар. Этот параметр можно задать только при запуске сервера.

По истечении этого времени старые данные могут быть вычищены. Это предотвращает замусоривание данными снимков, которые остаются задействованными долгое время. Во избежание получения некорректных результатов из-за очистки данных, которые должны были бы наблюдаться в снимке, клиенту будет выдана ошибка, если возраст снимка превысит заданный предел и из этого снимка будет запрошена страница, изменённая со времени его создания.

Значение -1 (по умолчанию) отключает это поведение. Полезные значения для производственной среды могут лежать в интервале от нескольких часов до нескольких дней. Заданное значение округляется до минут, а минимальные значения (как например, 0 или 1min) допускаются только потому, что они могут быть полезны при тестировании. Хотя допустимым будет и значение 60d (60 дней), учтите, что при многих видах нагрузки критичное замусоривание базы или зацикливание идентификаторов транзакций может происходить в намного меньших временных отрезках.

Когда это ограничение действует, освобождённое пространство в конце отношения не может быть отдано операционной системе, так как при этом будет удалена информация, необходимая для выявления условия снимок слишком стар. Всё пространство, выделенное отношению, останется связанным с ним до тех пор, пока не будет освобождено явно (например, с помощью команды VACUUM FULL).

Установка этого параметра не гарантирует, что обозначенная ошибка будет выдаваться при всех возможных обстоятельствах. На самом деле, если можно получить корректные результаты, например, из курсора, материализовавшего результирующий набор, ошибка не будет выдана, даже если нижележащие строки в целевой таблице были ликвидированы при очистке. Некоторые таблицы, например системные каталоги, не могут быть безопасно очищены в сжатые сроки, так что на них этот параметр не распространяется. Для таких таблиц этот параметр не сокращает раздувание, но и не чреват ошибкой снимок слишком стар при сканировании.

18.5. Журнал предзаписи

За дополнительной информацией о настройке этих параметров обратитесь к Разделу 29.4.

18.5.1. Параметры

wal_level (enum)

Параметр wal_level определяет, как много информации записывается в WAL. Со значением replica (по умолчанию) в журнал записываются данные, необходимые для поддержки архивирования WAL и репликации, включая запросы только на чтение на ведомом сервере. Вариант minimal оставляет только информацию, необходимую для восстановления после сбоя или аварийного отключения. Наконец, logical добавляет информацию, требующуюся для поддержки логического декодирования. Каждый последующий уровень включает информацию, записываемую на всех уровнях ниже. Задать этот параметр можно только при запуске сервера.

На уровне minimal некоторые массовые операции могут выполняться в обход журнала без риска потери данных, и при этом они выполнятся гораздо быстрее (см. Подраздел 14.4.7). В частности, такая оптимизация возможна с операциями:

CREATE TABLE AS

CREATE INDEX
CLUSTER
COPY с таблицами, которые были созданы или опустошены в той же транзакции

Однако такой минимальный журнал не будет содержать достаточно информации для восстановления данных из базовой копии и журналов, поэтому для реализации стратегии архивации WAL (см. archive_mode) и потоковой репликации необходим уровень replica или более высокий.

На уровне logical в журнал записывается та же информация, что и на уровне replica, плюс информация, необходимая для извлечения из журнала наборов логических изменений. Повышение уровня до logical приводит к значительному увеличению объёма WAL, особенно если многие таблицы имеют характеристику REPLICA IDENTITY FULL и выполняется множество команд UPDATE и DELETE.

В выпусках до 9.6 для этого параметра допускались значения archive и hot_standby. Эти значения по-прежнему принимаются, но теперь отображаются в значение replica.

fsync (boolean)

Если этот параметр установлен, сервер Postgres Pro старается добиться, чтобы изменения были записаны на диск физически, выполняя системные вызовы fsync() или другими подобными методами (см. wal_sync_method). Это даёт гарантию, что кластер баз данных сможет вернуться в согласованное состояние после сбоя оборудования или операционной системы.

Хотя отключение fsync часто даёт выигрыш в скорости, это может привести к неисправимой порче данных в случае отключения питания или сбоя системы. Поэтому отключать fsync рекомендуется, только если вы легко сможете восстановить всю базу из внешнего источника.

В качестве примеров, когда отключение fsync неопасно, можно привести начальное наполнение нового кластера данными из копии, обработку массива данных, после которой базу данных можно удалить и создать заново, либо эксплуатацию копии базы данных только для чтения, которая регулярно пересоздаётся и не используется для отработки отказа. Качественное оборудование само по себе не является достаточной причиной для отключения fsync.

При смене значения fsync с off на on для надёжного восстановления также необходимо сбросить все изменённые буферы из ядра в надёжное хранилище. Это можно сделать, когда сервер остановлен или когда режим fsync включён, с помощью команды initdb --sync-only, либо выполнить команду sync, размонтировать файловую систему или перезагрузить сервер.

Во многих случаях отключение synchronous_commit для некритичных транзакций может дать больший выигрыш в скорости, чем отключение fsync, при этом не добавляя риски повреждения данных.

Параметр fsync можно задать только в файле postgresql.conf или в командной строке при запуске сервера. Если вы отключаете этот параметр, возможно, имеет смысл отключить также и full_page_writes.

synchronous_commit (enum)

Определяет, после завершения какого уровня обработки WAL сервер будет сообщать об успешном выполнении операции. Допустимые значения: remote_apply (применено удалённо), on (вкл., по умолчанию), remote_write (записано удалённо), local (локально) и off (выкл.).

Если значение synchronous_standby_names не задано, для данного параметра имеют смысл только значения on и off; с вариантами remote_apply, remote_write и local будет выбран тот же уровень синхронизации, что и с on. Локальное действие всех отличных от off режимов заключается в ожидании локального сброса WAL на диск. В режиме off ожидание отсутствует, поэтому может образоваться окно от момента, когда клиент узнаёт об успешном

завершении, до момента, когда транзакция действительно гарантированно защищена от сбоя. (Максимальный размер окна равен тройному значению wal_writer_delay.) В отличие от fsync, значение `off` этого параметра не угрожает целостности данных: сбой операционной системы или базы данных может привести к потере последних транзакций, считавшихся зафиксированными, но состояние базы данных будет точно таким же, как и в случае штатного прерывания этих транзакций. Поэтому выключение режима `synchronous_commit` может быть полезной альтернативой отключению fsync, когда производительность важнее, чем надёжная гарантия сохранности каждой транзакции. Подробнее это обсуждается в Разделе 29.3.

Если значение synchronous_standby_names не пустое, параметр `synchronous_commit` также определяет, должен ли сервер при фиксировании транзакции ждать, пока соответствующие записи WAL будут обработаны на ведомом сервере (серверах).

Со значением `remote_apply` фиксирование завершается только после получения ответов от текущих синхронных ведомых серверов, говорящих, что они получили запись о фиксировании транзакции, сохранили её в надёжном хранилище, а также применили транзакцию, так что она стала видна для запросов на этих серверах. С таким вариантом задержка при фиксировании оказывается больше, так как необходимо дожидаться воспроизведения WAL. Со значением `on` фиксирование завершается только после получения ответов от текущих синхронных ведомых серверов, подтверждающих, что они получили запись о фиксировании транзакции и передали её в надёжном хранилище. Это гарантирует, что транзакция не будет потеряна, если только база данных не будет повреждена и на ведущем, и на всех синхронных ведомых серверах. Со значением `remote_write` фиксирование завершается после получения ответов от текущих синхронных серверов, говорящих, что они получили запись о фиксировании транзакции и сохранили её в своих ФС. Этот вариант позволяет гарантировать сохранность данных в случае отказа ведомого сервера Postgres Pro, но не в случае сбоя на уровне ОС, так как данные могут ещё не достичь надёжного хранилища на этом сервере. Со значением `local` фиксирование завершается после локального сброса данных, не дожидаясь репликации. Обычно это нежелательный вариант при синхронной репликации, но он представлен для полноты.

Этот параметр можно изменить в любое время; поведение каждой конкретной транзакции определяется значением, действующим в момент её фиксирования. Таким образом, есть возможность и смысл фиксировать некоторые транзакции синхронно, а другие — асинхронно. Например, чтобы зафиксировать одну транзакцию из нескольких команд асинхронно, когда по умолчанию выбран противоположный вариант, выполните в этой транзакции `SET LOCAL synchronous_commit TO OFF`.

Характеристики различных значений `synchronous_commit` сведены в Таблице 18.1.

Таблица 18.1. Режимы synchronous_commit

значение synchronous_commit	гарантированная локальная фиксация	гарантированная фиксация на ведомом после сбоя PG	гарантированная фиксация на ведомом после сбоя ОС	согласованность запросов на ведомом
remote_apply	•	•	•	•
on	•	•	•	
remote_write	•	•		
local	•			
off				

`wal_sync_method` (`enum`)

Метод, применяемый для принудительного сохранения изменений WAL на диске. Если режим `fsync` отключён, данный параметр не действует, так как принудительное сохранение изменений WAL не производится вовсе. Возможные значения этого параметра:

- open_datasync (для сохранения файлов WAL открывать их функцией open() с параметром O_DSYNC)
- fdatasync (вызывать fdatasync() при каждом фиксировании)
- fsync (вызывать fsync() при каждом фиксировании)
- fsync_writethrough (вызывать fsync() при каждом фиксировании, форсируя сквозную запись кеша)
- open_sync (для сохранения файлов WAL открывать их функцией open() с параметром O_SYNC)

Варианты open_* также применяют флаг O_DIRECT, если он доступен. Не все эти методы поддерживается в разных системах. По умолчанию выбирается первый из этих методов, который поддерживается текущей системой, с одним исключением — в Linux и FreeBSD по умолчанию выбирается fdatasync. Выбираемый по умолчанию вариант не обязательно будет идеальным; в зависимости от требований к отказоустойчивости или производительности может потребоваться скорректировать выбранное значение или внести другие изменения в конфигурацию вашей системы. Соответствующие аспекты конфигурации рассматриваются в Разделе 29.1. Этот параметр можно задать только в файле postgresql.conf или в командной строке при запуске сервера.

full_page_writes (boolean)

Когда этот параметр включён, сервер Postgres Pro записывает в WAL всё содержимое каждой страницы при первом изменении этой страницы после контрольной точки. Это необходимо, потому что запись страницы, прерванная при сбое операционной системы, может выполниться частично, и на диске окажется страница, содержащая смесь старых данных с новыми. При этом информации об изменениях на уровне строк, которая обычно сохраняется в WAL, будет недостаточно для получения согласованного содержимого такой страницы при восстановлении после сбоя. Сохранение образа всей страницы гарантирует, что страницу можно восстановить корректно, ценой увеличения объёма данных, которые будут записываться в WAL. (Так как воспроизведение WAL всегда начинается от контрольной точки, достаточно сделать это при первом изменении каждой страницы после контрольной точки. Таким образом, уменьшить затраты на запись полных страниц можно, увеличив интервалы контрольных точек.)

Отключение этого параметра ускоряет обычные операции, но может привести к неисправимому повреждению или незаметной порче данных после сбоя системы. Так как при этом возникают практически те же риски, что и при отключении fsync, хотя и в меньшей степени, отключать его следует только при тех же обстоятельствах, которые перечислялись в рекомендациях для вышеописанного параметра.

Отключение этого параметра не влияет на возможность применения архивов WAL для восстановления состояния на момент времени (см. Раздел 24.3).

Этот параметр можно задать только в postgresql.conf или в командной строке при запуске сервера. По умолчанию этот параметр имеет значение on.

wal_log_hints (boolean)

Когда этот параметр имеет значение on, сервер Postgres Pro записывает в WAL всё содержимое каждой страницы при первом изменении этой страницы после контрольной точки, даже при второстепенных изменениях так называемых вспомогательных битов.

Если включён расчёт контрольных сумм данных, изменения вспомогательных битов всегда проходят через WAL и этот параметр игнорируется. С помощью этого параметра можно проверить, насколько больше дополнительной информации записывалось бы в журнал, если бы для базы данных был включён подсчёт контрольных сумм.

Этот параметр можно задать только при запуске сервера. По умолчанию он имеет значение off.

`wal_compression` (`boolean`)

Когда этот параметр имеет значение `on`, сервер Postgres Pro сжимает образ полной страницы, записываемый в WAL, когда включён режим full_page_writes или при создании базовой копии. Сжатый образ страницы будет развёрнут при воспроизведении WAL. Значение по умолчанию — `off`. Изменить этот параметр могут только суперпользователи.

Этот параметр позволяет без дополнительных рисков повреждения данных уменьшить объём WAL, ценой дополнительной нагрузки на процессор, связанной со сжатием данных при записи в WAL и разворачиванием их при воспроизведении WAL.

`wal_buffers` (`integer`)

Объём разделяемой памяти, который будет использоваться для буферизации данных WAL, ещё не записанных на диск. Значение по умолчанию, равное -1, задаёт размер, равный 1/32 (около 3%) от shared_buffers, но не меньше чем `64 КБ` и не больше чем размер одного сегмента WAL (обычно `16 МБ`). Это значение можно задать вручную, если выбираемое автоматически слишком мало или велико, но при этом любое положительное число меньше `32 КБ` будет восприниматься как `32 КБ`. Этот параметр можно задать только при запуске сервера.

Содержимое буферов WAL записывается на диск при фиксировании каждой транзакции, так что очень большие значения вряд ли принесут значительную пользу. Однако значение как минимум в несколько мегабайт может увеличить быстродействие при записи на нагруженном сервере, когда сразу множество клиентов фиксируют транзакции. Автонастройка, действующая при значении по умолчанию (-1), в большинстве случаев выбирает разумные значения.

`wal_writer_delay` (`integer`)

Определяет, как часто процесс записи WAL будет сбрасывать WAL на диск. После очередного сброса WAL он делает паузу на `wal_writer_delay` миллисекунд, но может быть пробуждён асинхронно фиксируемой транзакцией. Если предыдущая операция сброса имела место в последние `wal_writer_delay` миллисекунд и после неё было получено меньше `wal_writer_flush_after` байт WAL, данные WAL только передаются ОС, но не сбрасываются на диск. Значение по умолчанию — 200 миллисекунд (`200ms`). Заметьте, что во многих системах разрешение таймера паузы составляет 10 мс; если задать в `wal_writer_delay` значение, не кратное 10, может быть получен тот же результат, что и со следующим за ним кратным 10. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`wal_writer_flush_after` (`integer`)

Определяет, как часто процесс записи WAL будет сбрасывать WAL на диск. Если предыдущая операция сброса имела место в последние `wal_writer_delay` миллисекунд и за это время было получено меньше `wal_writer_flush_after` байт WAL, данные WAL только передаются операционной системе, но не сбрасываются на диск. Если `wal_writer_flush_after` равен `0`, WAL сбрасывается на диск немедленно. Значение по умолчанию — 1 мегабайт (`1MB`). Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`commit_delay` (`integer`)

Параметр `commit_delay` добавляет паузу (в микросекундах) перед собственно выполнением сохранения WAL. Эта задержка может увеличить быстродействие при фиксировании множества транзакций, позволяя зафиксировать большее число транзакций за одну операцию сохранения WAL, если система нагружена достаточно сильно и за заданное время успевают зафиксироваться другие транзакции. Однако этот параметр также увеличивает задержку максимум до `commit_delay` микросекунд при каждом сохранении WAL. Эта задержка окажется бесполезной, если никакие другие транзакции не будут зафиксированы за это время, поэтому она добавляется, только если в момент запроса сохранения WAL активны как минимум `commit_siblings` других транзакций. Кроме того, эти задержки не добавляются

при выключенном fsync. По умолчанию значение commit_delay равно нулю (задержка отсутствует). Изменить этот параметр могут только суперпользователи.

В PostgreSQL до версии 9.3, параметр commit_delay работал по-другому и не так эффективно: он задерживал только фиксирование транзакций, а не все операции сохранения WAL, и заданная пауза выдерживалась полностью, даже если WAL удавалось сохранить быстрее. Начиная с версии 9.3, заданное время ожидает только первый процесс, готовый произвести сохранение, тогда как все последующие процессы ждут только, когда он закончит эту операцию.

commit_siblings (integer)

Минимальное число одновременно открытых транзакций, при котором будет добавляться задержка commit_delay. Чем больше это значение, тем больше вероятность, что минимум одна транзакция окажется готовой к фиксированию за время задержки. По умолчанию это число равно пяти.

18.5.2. Контрольные точки

checkpoint_timeout (integer)

Максимальное время между автоматическими контрольными точками в WAL (в секундах). Допускаются значения от 30 секунд до одного дня. Значение по умолчанию — пять минут (5min). Увеличение этого параметра может привести к увеличению времени, которое потребуется для восстановления после сбоя. Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера.

checkpoint_completion_target (floating point)

Задаёт целевое время для завершения процедуры контрольной точки, как коэффициент для общего времени между контрольными точками. По умолчанию это значение равно 0.5. Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера.

checkpoint_flush_after (integer)

Когда в процессе контрольной точки записывается больше чем checkpoint_flush_after байт, сервер даёт указание ОС произвести запись этих данных в нижележащее хранилище. Это ограничивает объём «грязных» данных в страничном кеше ядра и уменьшает вероятность затормаживания при выполнении fsync в конце этой контрольной точки или когда ОС сбрасывает данные на диск большими блоками в фоне. Часто это значительно уменьшает задержки транзакций, но бывают ситуации (особенно когда объём рабочей нагрузки больше shared_buffers, но меньше страничного кеша ОС), когда производительность может упасть. Этот параметр действует не на всех платформах. Он может принимать значение от 0 (при этом управление отложенной записью отключается) до 2 мегабайт (2MB). Значение по умолчанию — 256kB в Linux и 0 в других ОС. (Если BLCKSZ отличен от 8 КБ, значение по умолчанию и максимум корректируются пропорционально.) Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера.

checkpoint_warning (integer)

Записывать в журнал сервера сообщение в случае, если контрольные точки, вызванные заполнением файлов сегментов, выполняются раньше, чем через заданное число секунд (что говорит о том, что значение max_wal_size нужно увеличить). Значение по умолчанию равно 30 секундам (30s). При нуле это предупреждение отключается. Если checkpoint_timeout меньше, чем checkpoint_warning, предупреждения так же не будут выводиться. Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера.

max_wal_size (integer)

Максимальный размер, до которого может вырастать WAL во время автоматических контрольных точек. Это мягкий предел; размер WAL может превышать max_wal_size при особых обстоятельствах, например, при высокой нагрузке, сбое в archive_command или при большом значении wal_keep_segments. Значение по умолчанию — 1 ГБ. Увеличение этого

параметра может привести к увеличению времени, которое потребуется для восстановления после сбоя. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`min_wal_size` (`integer`)

Пока WAL занимает на диске меньше этого объёма, старые файлы WAL в контрольных точках всегда перерабатываются, а не удаляются. Это позволяет зарезервировать достаточно места для WAL, чтобы справиться с резкими скачками использования WAL, например, при выполнении больших пакетных заданий. Значение по умолчанию — 80 МБ. Этот параметр можно установить только в `postgresql.conf` или в командной строке сервера.

18.5.3. Архивация

`archive_mode` (`enum`)

Когда параметр `archive_mode` включён, полные сегменты WAL передаются в хранилище архива командой archive_command. Помимо значения `off` (выключающего архивацию) есть ещё два: `on` (вкл.) и `always` (всегда). В обычном состоянии эти два режима не различаются, но в режиме `always` архивация WAL активна и во время восстановления архива, и при использовании ведомого сервера. В этом режиме все файлы, восстановленные из архива или полученные при потоковой репликации, будут архивироваться (снова). За подробностями обратитесь к Подразделу 25.2.9.

Параметры `archive_mode` и `archive_command` разделены, чтобы команду архивации (`archive_command`) можно было изменять, не отключая режим архивации. Этот параметр можно задать только при запуске сервера. Режим архивации нельзя включить, когда установлен минимальный уровень WAL (`wal_level` имеет значение `minimal`).

`archive_command` (`string`)

Команда локальной оболочки, которая будет выполняться для архивации завершённого сегмента WAL. Любое вхождение `%p` в этой строке заменяется путём архивируемого файла, а вхождение `%f` заменяется только его именем. (Путь задаётся относительно рабочего каталога сервера, то есть каталога данных кластера.) Чтобы вставить в команду символ `%`, его нужно записать как `%%`. Важно, чтобы команда возвращала нулевой код, только если она завершается успешно. За дополнительной информацией обратитесь к Подразделу 24.3.1.

Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. Если режим архивации (`archive_mode`) не был включён при запуске, этот параметр игнорируется. Если значение `archive_command` — пустая строка (по умолчанию), но `archive_mode` включён, архивация WAL временно отключается, но сервер продолжает накапливать файлы сегментов WAL в ожидании, что команда будет вскоре определена. Если в качестве `archive_command` задать команду, которая ничего не делает, но сообщает об успешном завершении, например `/bin/true` (или `REM` в Windows), архивация по сути отключается, но при этом нарушается цепочка файлов WAL, необходимых для восстановления архива, поэтому такой вариант следует использовать только в особых случаях.

`archive_timeout` (`integer`)

Команда archive_command вызывается только для завершённых сегментов WAL. Поэтому, если ваш сервер записывает мало данных WAL (или это наблюдается в некоторые периоды времени), от завершения транзакции до надёжного сохранения её в архивном хранилище может пройти довольно много времени. Для ограничения времени существования неархивированных данных можно установить значение `archive_timeout`, чтобы сервер периодически переключался на новый файл сегмента WAL. Когда этот параметр больше нуля, сервер будет переключаться на новый файл сегмента, если с момента последнего переключения на новый файл прошло заданное число секунд, и наблюдалась какая-то активность базы данных, даже если это была просто контрольная точка. (Контрольные точки пропускаются, если в базе отсутствует активность). Заметьте, что архивируемые файлы, закрываемые досрочно из-за принудительного переключения, всё равно будут иметь тот же размер, что и полностью

заполненные. Поэтому устанавливать для archive_timeout очень маленькое значение неразумно — это ведёт к замусориванию архивного хранилища. Обычно для archive_timeout имеет смысл задавать значение около минуты. Если вам нужно, чтобы данные копировались с главного сервера быстрее, вам следует подумать о переходе от архивации к потоковой репликации. Этот параметр можно задать только в postgresql.conf или в командной строке при запуске сервера.

18.6. Репликация

Эти параметры управляют поведением встроенного механизма *потоковой репликации* (см. Подраздел 25.2.5). Когда он применяется, один сервер является ведущим, а другие — ведомыми. Ведущий сервер всегда передаёт данные, а ведомые всегда принимают данные репликации, но когда настроена каскадная репликация (см. Подраздел 25.2.7), ведомые серверы также могут быть и передающими. Большинство следующих параметров относится к передающим и ведомым серверам, но есть несколько параметров, которые имеют смысл только для ведущего. Все эти параметры вполне могут быть разными в рамках одного кластера, если это требуется.

18.6.1. Передающие серверы

Эти параметры можно задать на любом сервере, который передаёт данные репликации одному или нескольким ведомым. Ведущий сервер всегда является передающим, так что на нём они должны задаваться всегда. Роль и значение этих параметров не меняются после того, как ведомый сервер становится ведущим.

max_wal_senders (integer)

Задаёт максимально допустимое число одновременных подключений ведомых серверов или клиентов потокового копирования (т. е. максимальное количество одновременно работающих процессов передачи WAL). Значение по умолчанию — 10. Нулевое значение отключает репликацию. Процессы-передатчики WAL учитываются в общем числе соединений, так что этот параметр не может превышать max_connections. В случае неожиданного отключения клиента потоковой передачи слот подключения может оставаться занятым до достижения тайм-аута, так что этот параметр должен быть немного больше максимально допустимого числа клиентов, чтобы отключившиеся клиенты могли переподключиться немедленно. Задать этот параметр можно только при запуске сервера. Чтобы к данному серверу могли подключаться ведомые, уровень wal_level должен быть replica или выше.

max_replication_slots (integer)

Задаёт максимальное число слотов репликации (см. Подраздел 25.2.6), которое сможет поддерживать сервер. Значение по умолчанию — 10. Этот параметр можно задать только при запуске сервера. Чтобы эти слоты репликации можно было использовать, уровень wal_level должен быть replica или выше. Если заданное значение данного параметра будет меньше, чем число уже существующих слотов репликации, сервер не запустится.

На стороне подписчика указывает, сколько источников репликации (см. Главу 47) можно отслеживать одновременно, по сути ограничивая количество подписок на логическую репликацию, которые могут быть созданы на сервере. Если установленное значение будет меньше, чем текущее количество отслеживаемых источников репликации (показываемое в pg_replication_origin_status, а не в pg_replication_origin), сервер не запустится.

wal_keep_segments (integer)

Задаёт минимальное число файлов прошлых сегментов журнала, которые будут сохраняться в каталоге pg_wal, чтобы ведомый сервер мог выбрать их при потоковой репликации. Обычно сегмент имеет размер 16 мегабайт. Если ведомый сервер, подключённый к передающему, отстаёт больше чем на wal_keep_segments сегментов, передающий удаляет сегменты WAL, всё ещё необходимые ведомому, и в этом случае соединение репликации прерывается. В результате этого затем также будут прерваны зависимые соединения. (Однако ведомый сервер сможет восстановиться, выбрав этот сегмент из архива, если осуществляется архивация WAL.)

Этот параметр задаёт только минимальное число сегментов, сохраняемое в каталоге pg_wal; система может сохранить больше сегментов для архивации WAL или для восстановления с момента контрольной точки. Если wal_keep_segments равен нулю (это значение по умолчанию), система не сохраняет никакие дополнительные сегменты для ведомых серверов, поэтому число старых сегментов WAL, доступных для ведомых, зависит от положения предыдущей контрольной точки и состояния архивации WAL. Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера.

wal_sender_timeout (integer)

Задаёт период времени (в миллисекундах), по истечении которого прерываются неактивные соединения репликации. Это помогает передающему серверу обнаружить сбой ведомого или разрывы сети. При значении, равном нулю, тайм-аут отключается. Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера. Значение по умолчанию — 60 секунд.

track_commit_timestamp (boolean)

Включает запись времени фиксации транзакций. Этот параметр можно задать только в postgresql.conf или в командной строке при запуске сервера. По умолчанию этот параметр имеет значение off.

18.6.2. Главный сервер

Эти параметры можно задать на главном/ведущем сервере, который должен передавать данные репликации одному или нескольким ведомым. Заметьте, что помимо этих параметров на ведущем сервере должен быть правильно установлен wal_level, а также может быть включена архивация WAL (см. Подраздел 18.5.3). Значения этих параметров на ведомых серверах не важны, хотя их можно подготовить заранее, на случай, если ведомый сервер придётся сделать ведущим.

synchronous_standby_names (string)

Определяет список ведомых серверов, которые могут поддерживать *синхронную репликацию*, как описано в Подразделе 25.2.8. Активных синхронных ведомых серверов может быть один или несколько; транзакции, ожидающие фиксации, будут завершаться только после того, как эти ведомые подтвердят получение их данных. Синхронными ведомыми будут те, имена которых указаны в этом списке и которые подключены к ведущему и принимают поток данных в реальном времени (что показывает признак streaming в представлении pg_stat_replication). Указание нескольких имён ведомых серверов позволяет обеспечить очень высокую степень доступности и защиту от потери данных.

Именем ведомого сервера в этом контексте считается значение application_name этого сервера, задаваемое в свойствах подключения. При организации физической репликации оно задаётся в строке primary_conninfo в recovery.conf (по умолчанию — walreceiver). Для логической репликации его можно задать в строке подключения для подписки (по умолчанию это имя подписки). Как задать его для других потребителей потоков репликации, вы можете узнать в их документации.

Этот параметр принимает список ведомых серверов в одной из следующих форм:

```
[FIRST] число_синхронных ( имя_ведомого [, ...] )
ANY число_синхронных ( имя_ведомого [, ...] )
имя_ведомого [, ...]
```

здесь *число_синхронных* — число синхронных ведомых серверов, от которых необходимо дожидаться ответов для завершения транзакций, а *имя_ведомого* — имя ведомого сервера. Слова FIRST и ANY задают метод выбора синхронных ведомых из перечисленных серверов.

Ключевое слово FIRST, в сочетании с *числом_синхронных*, выбирает синхронную репликацию на основе приоритетов, когда транзакции фиксируются только после того, как их записи в WAL реплицируются на *число_синхронных* ведомых серверов, выбираемых согласно приоритетам. Например, со значением FIRST 3 (s1, s2, s3, s4) для фиксации транзакции необходимо

дождаться ответа от трёх наиболее приоритетных из серверов `s1`, `s2`, `s3` и `s4`. Ведомые серверы, имена которых идут в этом списке первыми, будут иметь больший приоритет и будут считаться синхронными. Серверы, следующие в списке за ними, будут считаться потенциальными синхронными. Если один из текущих синхронных серверов по какой-то причине отключается, он немедленно будет заменён следующим сервером с наибольшим приоритетом. Ключевое слово `FIRST` может быть опущено.

Ключевое слово `ANY`, в сочетании с *числом_синхронных*, выбирает синхронную репликацию на основе кворума, когда транзакции фиксируются только после того, как их записи в WAL реплицируются на *как минимум* *число_синхронных* перечисленных серверов. Например, со значением `ANY 3 (s1, s2, s3, s4)` для фиксации транзакции необходимо дождаться ответа от как минимум трёх из серверов `s1`, `s2`, `s3` и `s4`.

Ключевые слова `FIRST` и `ANY` воспринимаются без учёта регистра. Если такое же имя оказывается у одного из ведомых серверов, его *имя_ведомого* нужно заключить в двойные кавычки.

Третья форма использовалась в Postgres Pro до версии 9.6 и по-прежнему поддерживается. По сути она равнозначна первой с `FIRST` и *числом_синхронным*, равным 1. Например, `FIRST 1 (s1, s2)` и `s1, s2` действуют одинаково: в качестве синхронного ведомого выбирается либо `s1`, либо `s2`.

Специальному элементу `*` соответствует имя любого ведомого.

Уникальность имён ведомых серверов не контролируется. В случае дублирования имён более приоритетным будет один из серверов с подходящим именем, хотя какой именно, не определено.

> **Примечание**
>
> Каждое *имя_ведомого* должно задаваться в виде допустимого идентификатора SQL, кроме `*`. При необходимости его можно заключать в кавычки. Но заметьте, что идентификаторы *имя_ведомого* сравниваются с именами приложений без учёта регистра, независимо от того, заключены ли они в кавычки или нет.

Если имена синхронных ведомых серверов не определены, синхронная репликация не включается и фиксируемые транзакции не будут ждать репликации. Это поведение по умолчанию. Даже когда синхронная репликация включена, для отдельных транзакций можно отключить ожидание репликации, задав для параметра synchronous_commit значение `local` или `off`.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`vacuum_defer_cleanup_age` (`integer`)

Задаёт число транзакций, на которое будет отложена очистка старых версий строк при `VACUUM` и изменениях HOT. По умолчанию это число равно нулю, то есть старые версии строк могут удаляться сразу, как только перестанут быть видимыми в открытых транзакциях. Это значение можно сделать ненулевым на ведущем сервере, работающим с серверами горячего резерва, как описано в Разделе 25.5. В результате увеличится время, в течение которого будут успешно выполняться запросы на ведомом сервере без конфликтов из-за ранней очистки строк. Однако ввиду того, что эта отсрочка определяется числом записывающих транзакций, выполняющихся на ведущем сервере, сложно предсказать, каким будет дополнительное время отсрочки на ведомом сервере. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

В качестве альтернативы этому параметру можно также рассмотреть `hot_standby_feedback` на ведомом сервере.

Этот параметр не предотвращает очистку старых строк, которые достигли возраста, заданного параметром old_snapshot_threshold.

18.6.3. Ведомые серверы

Эти параметры управляют поведением ведомого сервера, который будет получать данные репликации. На ведущем сервере они не играют никакой роли.

hot_standby (boolean)

Определяет, можно ли будет подключаться к серверу и выполнять запросы в процессе восстановления, как описано в Разделе 25.5. Значение по умолчанию — on (подключения разрешаются). Задать этот параметр можно только при запуске сервера. Данный параметр играет роль только в режиме ведомого сервера или при восстановлении архива.

max_standby_archive_delay (integer)

В режиме горячего резерва этот параметр определяет, как долго должен ждать ведомый сервер, прежде чем отменять запросы, конфликтующие с очередными изменениями в WAL, как описано в Подразделе 25.5.2. Задержка max_standby_archive_delay применяется при обработке данных WAL, считываемых из архива (не текущих данных). Значение этого параметра задаётся в миллисекундах (если явно не указаны другие единицы) и по умолчанию равно 30 секундам. При значении, равном -1, ведомый может ждать завершения конфликтующих запросов неограниченное время. Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера.

Заметьте, что параметр max_standby_archive_delay определяет не максимальное время, которое отводится для выполнения каждого запроса, а максимальное общее время, за которое должны быть применены изменения из одного сегмента WAL. Таким образом, если один запрос привёл к значительной задержке при обработке сегмента WAL, остальным конфликтующим запросам будет отведено гораздо меньше времени.

max_standby_streaming_delay (integer)

В режиме горячего резерва этот параметр определяет, как долго должен ждать ведомый сервер, прежде чем отменять запросы, конфликтующие с очередными изменениями в WAL, как описано в Подразделе 25.5.2. Задержка max_standby_streaming_delay применяется при обработке данных WAL, поступающих при потоковой репликации. Значение этого параметра задаётся в миллисекундах (если явно не указаны другие единицы) и по умолчанию равно 30 секундам. При значении, равном -1, ведомый может ждать завершения конфликтующих запросов неограниченное время. Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера.

Заметьте, что параметр max_standby_streaming_delay определяет не максимальное время, которое отводится для выполнения каждого запроса, а максимальное общее время, за которое должны быть применены изменения из WAL после получения от главного сервера. Таким образом, если один запрос привёл к значительной задержке, остальным конфликтующим запросам будет отводиться гораздо меньше времени, пока резервный сервер не догонит главный.

wal_receiver_status_interval (integer)

Определяет минимальную частоту, с которой процесс, принимающий WAL на ведомом сервере, будет сообщать о состоянии репликации ведущему или вышестоящему ведомому, где это состояние можно наблюдать в представлении pg_stat_replication. В этом сообщении передаются следующие позиции в журнале предзаписи: позиция изменений записанных, изменений, сохранённых на диске, и изменений применённых. Значение параметра задаётся в секундах и определяет максимальный интервал между сообщениями. Сообщения о состоянии передаются при каждом продвижении позиций записанных или сохранённых на диске изменений, но с промежутком не больше, чем заданный этим параметром. Таким образом, последняя переданная позиция применённых изменений может немного отставать от

фактической в текущий момент. При нулевом значении этого параметра передача состояния полностью отключается. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера. По умолчанию его значение равно 10 секундам.

`hot_standby_feedback` (`boolean`)

Определяет, будет ли сервер горячего резерва сообщать ведущему или вышестоящему ведомому о запросах, которые он выполняет в данный момент. Это позволяет исключить необходимость отмены запросов, вызванную очисткой записей, но при некоторых типах нагрузки это может приводить к раздуванию базы данных на ведущем сервере. Эти сообщения о запросах будут отправляться не чаще, чем раз в интервал, задаваемый параметром `wal_receiver_status_interval`. Значение данного параметра по умолчанию — `off`. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

Если используется каскадная репликация, сообщения о запросах передаются выше, пока в итоге не достигнут ведущего сервера. На промежуточных серверах эта информация больше никак не задействуется.

Этот параметр не переопределяет поведение `old_snapshot_threshold`, установленное на ведущем сервере; снимок на ведомом сервере, имеющий возраст больше заданного указанным параметром на ведущем, может стать недействительным, что приведёт к отмене транзакций на ведомом. Это объясняется тем, что предназначение `old_snapshot_threshold` заключается в указании абсолютного ограничения времени, в течение которого могут накапливаться мёртвые строки, которое иначе могло бы нарушаться из-за конфигурации ведомого.

`wal_receiver_timeout` (`integer`)

Задаёт период времени (в миллисекундах), по истечении которого прерываются неактивные соединения репликации. Это помогает принимающему ведомому серверу обнаружить сбой ведущего или разрыв сети. При значении, равном нулю, тайм-аут отключается. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера. Значение по умолчанию — 60 секунд.

`wal_retrieve_retry_interval` (`integer`)

Определяет, сколько ведомый сервер должен ждать поступления данных WAL из любых источников (потоковая репликация, локальный `pg_wal` или архив WAL), прежде чем повторять попытку получения WAL. Задать этот параметр можно только в `postgresql.conf` или в командной строке сервера. Значение по умолчанию — 5 секунд. Если единицы не задаются, подразумеваются миллисекунды.

Этот параметр полезен в конфигурациях, когда для узла в схеме восстановления нужно регулировать время ожидания новых данных WAL. Например, при восстановлении архива можно ускорить реакцию на появление нового файла WAL, уменьшив значение этого параметра. В системе с низкой активностью WAL увеличение этого параметра приведёт к сокращению числа запросов, необходимых для отслеживания архивов WAL, что может быть полезно в облачных окружениях, где учитывается число обращений к инфраструктуре.

18.6.4. Подписчики

Эти параметры управляют поведением подписчика логической репликации. На публикующем сервере они не играют роли.

Заметьте, что параметры конфигурации `wal_receiver_timeout`, `wal_receiver_status_interval` и `wal_retrieve_retry_interval` также воздействуют на рабочие процессы логической репликации.

`max_logical_replication_workers` (`int`)

Задаёт максимально возможное число рабочих процессов логической репликации. В это число входят и рабочие процессы, применяющие изменения, и процессы, синхронизирующие таблицы.

Рабочие процессы логической репликации берутся из пула, контролируемого параметром max_worker_processes.

Значение по умолчанию — 4. Этот параметр можно задать только при запуске сервера.

max_sync_workers_per_subscription (integer)

Максимальное число рабочих процессов, выполняющих синхронизацию, для одной подписки. Этот параметр управляет степенью распараллеливания копирования начальных данных в процессе инициализации подписки или при добавлении новых таблиц.

В настоящее время одну таблицу может обрабатывать только один рабочий процесс синхронизации.

Рабочие процессы синхронизации берутся из пула, контролируемого параметром max_logical_replication_workers.

Значение по умолчанию — 2. Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера.

18.7. Планирование запросов

18.7.1. Конфигурация методов планировщика

Эти параметры конфигурации дают возможность грубо влиять на планы, выбираемые оптимизатором запросов. Если автоматически выбранный оптимизатором план конкретного запроса оказался неоптимальным, в качестве *временного* решения можно воспользоваться одним из этих параметров и вынудить планировщик выбрать другой план. Улучшить качество планов, выбираемых планировщиком, можно и более подходящими способами, в частности, скорректировать константы стоимости (см. Подраздел 18.7.2), выполнить ANALYZE вручную, изменить значение параметра конфигурации default_statistics_target и увеличить объём статистики, собираемой для отдельных столбцов, воспользовавшись командой ALTER TABLE SET STATISTICS.

enable_bitmapscan (boolean)

Включает или отключает использование планов сканирования по битовой карте. По умолчанию имеет значение on (вкл.).

enable_gathermerge (boolean)

Включает или отключает использование планов соединения посредством сбора. По умолчанию имеет значение on (вкл.).

enable_hashagg (boolean)

Включает или отключает использование планов агрегирования по хешу. По умолчанию имеет значение on (вкл.).

enable_hashjoin (boolean)

Включает или отключает использование планов соединения по хешу. По умолчанию имеет значение on (вкл.).

enable_indexscan (boolean)

Включает или отключает использование планов сканирования по индексу. По умолчанию имеет значение on (вкл.).

enable_indexonlyscan (boolean)

Включает или отключает использование планов сканирования только индекса (см. Раздел 11.11). По умолчанию имеет значение on (вкл.).

`enable_material` (`boolean`)

Включает или отключает использование материализации при планировании запросов. Полностью исключить материализацию невозможно, но при выключении этого параметра планировщик не будет вставлять узлы материализации, за исключением случаев, где они требуются для правильности. По умолчанию этот параметр имеет значение `on` (вкл.).

`enable_mergejoin` (`boolean`)

Включает или отключает использование планов соединения слиянием. По умолчанию имеет значение `on` (вкл.).

`enable_nestloop` (`boolean`)

Включает или отключает использование планировщиком планов соединения с вложенными циклами. Полностью исключить вложенные циклы невозможно, но при выключении этого параметра планировщик не будет использовать данный метод, если можно применить другие. По умолчанию этот параметр имеет значение `on`.

`enable_seqscan` (`boolean`)

Включает или отключает использование планировщиком планов последовательного сканирования. Полностью исключить последовательное сканирование невозможно, но при выключении этого параметра планировщик не будет использовать данный метод, если можно применить другие. По умолчанию этот параметр имеет значение `on`.

`enable_sort` (`boolean`)

Включает или отключает использование планировщиком шагов с явной сортировкой. Полностью исключить явную сортировку невозможно, но при выключении этого параметра планировщик не будет использовать данный метод, если можно применить другие. По умолчанию этот параметр имеет значение `on`.

`enable_tidscan` (`boolean`)

Включает или отключает использование планов сканирования TID. По умолчанию имеет значение `on` (вкл.).

18.7.2. Константы стоимости для планировщика

Переменные *стоимости*, описанные в данном разделе, задаются по произвольной шкале. Значение имеют только их отношения, поэтому умножение или деление всех переменных на один коэффициент никак не повлияет на выбор планировщика. По умолчанию эти переменные определяются относительно стоимости чтения последовательной страницы: то есть, переменную `seq_page_cost` удобно задать равной `1.0`, а все другие переменные стоимости определить относительно неё. Но при желании можно использовать и другую шкалу, например, выразить в миллисекундах фактическое время выполнения запросов на конкретной машине.

> **Примечание**
>
> К сожалению, какого-либо чётко определённого способа определения идеальных значений стоимости не существует. Лучше всего выбирать их как средние показатели при выполнении целого ряда разнообразных запросов, которые будет обрабатывать конкретная СУБД. Это значит, что менять их по результатам всего нескольких экспериментов очень рискованно.

`seq_page_cost` (`floating point`)

Задаёт приблизительную стоимость чтения одной страницы с диска, которое выполняется в серии последовательных чтений. Значение по умолчанию равно 1.0. Это значение можно переопределить для таблиц и индексов в определённом табличном пространстве, установив одноимённый параметр табличного пространства (см. ALTER TABLESPACE).

`random_page_cost` (`floating point`)

Задаёт приблизительную стоимость чтения одной произвольной страницы с диска. Значение по умолчанию равно 4.0. Это значение можно переопределить для таблиц и индексов в определённом табличном пространстве, установив одноимённый параметр табличного пространства (см. ALTER TABLESPACE).

При уменьшении этого значения по отношению к `seq_page_cost` система начинает предпочитать сканирование по индексу; при увеличении такое сканирование становится более дорогостоящим. Оба эти значения также можно увеличить или уменьшить одновременно, чтобы изменить стоимость операций ввода/вывода по отношению к стоимости процессорных операций, которая определяется следующими параметрами.

Произвольный доступ к механическому дисковому хранилищу обычно гораздо дороже последовательного доступа, более чем в четыре раза. Однако по умолчанию выбран небольшой коэффициент (4.0), в предположении, что большой объём данных при произвольном доступе, например, при чтении индекса, окажется в кеше. Таким образом, можно считать, что значение по умолчанию моделирует ситуацию, когда произвольный доступ в 40 раз медленнее последовательного, но 90% операций произвольного чтения удовлетворяются из кеша.

Если вы считаете, что для вашей рабочей нагрузки процент попаданий не достигает 90%, вы можете увеличить параметр random_page_cost, чтобы он больше соответствовал реальной стоимости произвольного чтения. И напротив, если ваши данные могут полностью поместиться в кеше, например, когда размер базы меньше общего объёма памяти сервера, может иметь смысл уменьшить random_page_cost. С хранилищем, у которого стоимость произвольного чтения не намного выше последовательного, как например, у твердотельных накопителей, так же лучше выбрать меньшее значение random_page_cost, например `1.1`.

> **Подсказка**
>
> Хотя система позволяет сделать `random_page_cost` меньше, чем `seq_page_cost`, это лишено физического смысла. Однако сделать их равными имеет смысл, если база данных полностью кешируется в ОЗУ, так как в этом случае с обращением к страницам в произвольном порядке не связаны никакие дополнительные издержки. Кроме того, для сильно загруженной базы данных оба этих параметра следует понизить по отношению к стоимости процессорных операций, так как стоимость выборки страницы, уже находящейся в ОЗУ, оказывается намного меньше, чем обычно.

`cpu_tuple_cost` (`floating point`)

Задаёт приблизительную стоимость обработки каждой строки при выполнении запроса. Значение по умолчанию — 0.01.

`cpu_index_tuple_cost` (`floating point`)

Задаёт приблизительную стоимость обработки каждой записи индекса при сканировании индекса. Значение по умолчанию — 0.005.

`cpu_operator_cost` (`floating point`)

Задаёт приблизительную стоимость обработки оператора или функции при выполнении запроса. Значение по умолчанию — 0.0025.

`parallel_setup_cost` (`floating point`)

Задаёт приблизительную стоимость запуска параллельных рабочих процессов. Значение по умолчанию — 1000.

`parallel_tuple_cost` (`floating point`)

Задаёт приблизительную стоимость передачи одного кортежа от параллельного рабочего процесса другому процессу. Значение по умолчанию — 0.1.

`min_parallel_table_scan_size` (`integer`)

Задаёт минимальный объём данных таблицы, подлежащий сканированию, при котором может применяться параллельное сканирование. Для параллельного последовательного сканирования объём сканируемых данных всегда равняется размеру таблицы, но когда используются индексы, этот объём обычно меньше. Значение по умолчанию — 8 мегабайт (`8MB`).

`min_parallel_index_scan_size` (`integer`)

Задаёт минимальный объём данных индекса, подлежащий сканированию, при котором может применяться параллельное сканирование. Заметьте, что при параллельном сканировании по индексу обычно не затрагивается весь индекс; здесь учитывается число страниц, которое по мнению планировщика будет затронуто при сканировании. Значение по умолчанию — 512 килобайт (`512kB`).

`effective_cache_size` (`integer`)

Определяет представление планировщика об эффективном размере дискового кеша, доступном для одного запроса. Это представление влияет на оценку стоимости использования индекса; чем выше это значение, тем больше вероятность, что будет применяться сканирование по индексу, чем ниже, тем более вероятно, что будет выбрано последовательное сканирование. При установке этого параметра следует учитывать и объём разделяемых буферов Postgres Pro, и процент дискового кеша ядра, который будут занимать файлы данных Postgres Pro, хотя некоторые данные могут оказаться и там, и там. Кроме того, следует принять во внимание ожидаемое число параллельных запросов к разным таблицам, так как общий размер будет разделяться между ними. Этот параметр не влияет на размер разделяемой памяти, выделяемой Postgres Pro, и не задаёт размер резервируемого в ядре дискового кеша; он используется только в качестве ориентировочной оценки. При этом система не учитывает, что данные могут оставаться в дисковом кеше от запроса к запросу. Значение этого параметра по умолчанию — 4 гигабайта (`4GB`).

18.7.3. Генетический оптимизатор запросов

Генетический оптимизатор запросов (GEnetic Query Optimizer, GEQO) осуществляет планирование запросов, применяя эвристический поиск. Это позволяет сократить время планирования для сложных запросов (в которых соединяются множество отношений), ценой того, что иногда полученные планы уступают по качеству планам, выбираемым при полном переборе. За дополнительными сведениями обратитесь к Главе 55.

`geqo` (`boolean`)

Включает или отключает генетическую оптимизацию запросов. По умолчанию она включена. В производственной среде её лучше не отключать; более гибко управлять GEQO можно с помощью переменной `geqo_threshold`.

`geqo_threshold` (`integer`)

Задаёт минимальное число элементов во `FROM`, при котором для планирования запроса будет привлечён генетический оптимизатор. (Заметьте, что конструкция `FULL OUTER JOIN` считается одним элементом списка `FROM`.) Значение по умолчанию — 12. Для более простых запросов часто лучше использовать обычный планировщик, производящий полный перебор, но для запросов со множеством таблиц полный перебор займёт слишком много времени, чаще гораздо больше, чем будет потеряно из-за выбора не самого эффективного плана. Таким образом, ограничение по размеру запроса даёт удобную возможность управлять GEQO.

`geqo_effort` (`integer`)

Управляет выбором между сокращением временем планирования и повышением качества плана запроса в GEQO. Это значение должна задаваться целым числом от 1 до 10. Значение по умолчанию равно пяти. Чем больше значение этого параметра, тем больше времени будет потрачено на планирование запроса, но и тем больше вероятность, что будет выбран эффективный план.

Параметр geqo_effort сам по себе ничего не делает, он используется только для вычисления значений по умолчанию для других переменных, влияющих на поведение GEQO (они описаны ниже). При желании эти переменные можно просто установить вручную.

geqo_pool_size (integer)

Задаёт размер пула для алгоритма GEQO, то есть число особей в генетической популяции. Это число должно быть не меньше двух, но полезные значения обычно лежат в интервале от 100 до 1000. Если оно равно нулю (это значение по умолчанию), то подходящее число выбирается, исходя из значения geqo_effort и числа таблиц в запросе.

geqo_generations (integer)

Задаёт число поколений для GEQO, то есть число итераций этого алгоритма. Оно должно быть не меньше единицы, но полезные значения находятся в том же диапазоне, что и размер пула. Если оно равно нулю (это значение по умолчанию), то подходящее число выбирается, исходя из geqo_pool_size.

geqo_selection_bias (floating point)

Задаёт интенсивность селекции для GEQO, то есть селективное давление в популяции. Допустимые значения лежат в диапазоне от 1.50 до 2.00 (это значение по умолчанию).

geqo_seed (floating point)

Задаёт начальное значение для генератора случайных чисел, который применяется в GEQO для выбора случайных путей в пространстве поиска порядка соединений. Может иметь значение от нуля (по умолчанию) до одного. При изменении этого значения меняется набор анализируемых путей, в результате чего может быть найден как более, так и менее оптимальный путь.

18.7.4. Другие параметры планировщика

default_statistics_target (integer)

Устанавливает значение ориентира статистики по умолчанию, распространяющееся на столбцы, для которых командой ALTER TABLE SET STATISTICS не заданы отдельные ограничения. Чем больше установленное значение, тем больше времени требуется для выполнения ANALYZE, но тем выше может быть качество оценок планировщика. Значение этого параметра по умолчанию — 100. За дополнительными сведениями об использовании статистики планировщиком запросов Postgres Pro обратитесь к Разделу 14.2.

constraint_exclusion (enum)

Управляет использованием ограничений таблиц для оптимизации запросов. Допустимые значения constraint_exclusion: on (задействовать ограничения всех таблиц), off (никогда не задействовать ограничения) и partition (задействовать ограничения только для дочерних таблиц и подзапросов UNION ALL). Значение по умолчанию — partition. Оно часто помогает увеличить производительность, когда применяются секционированные таблицы и наследование.

Когда данный параметр разрешает это для таблицы, планировщик сравнивает условия запроса с ограничениями CHECK данной таблицы и не сканирует её, если они оказываются несовместимыми. Например:

```
CREATE TABLE parent(key integer, ...);
CREATE TABLE child1000(check (key between 1000 and 1999)) INHERITS(parent);
CREATE TABLE child2000(check (key between 2000 and 2999)) INHERITS(parent);
...
SELECT * FROM parent WHERE key = 2400;
```

Если включено исключение по ограничению, команда SELECT не будет сканировать таблицу child1000, в результате чего запрос выполнится быстрее.

В настоящее время исключение по ограничению разрешено по умолчанию только в условиях, возникающих при реализации секционированных таблиц. Включение этой возможности для всех таблиц влечёт дополнительные издержки на планирование, довольно заметные для простых запросов, но никакого выигрыша это не приносит. Если вы не применяете секционированные таблицы, лучше всего полностью отключить эту возможность.

За дополнительными сведениями об исключении по ограничению и секционировании таблиц обратитесь к Подразделу 5.10.4.

`cursor_tuple_fraction` (`floating point`)

Задаёт для планировщика оценку процента строк, которые будут получены через курсор. Значение по умолчанию — 0.1 (10%). При меньших значениях планировщик будет склонен использовать для курсоров планы с «быстрым стартом», позволяющие получать первые несколько строк очень быстро, хотя для выборки всех строк может уйти больше времени. При больших значениях планировщик стремится оптимизировать общее время запроса. При максимальном значении, равном 1.0, работа с курсорами планируется так же, как и обычные запросы — минимизируется только общее время, а не время получения первых строк.

`from_collapse_limit` (`integer`)

Задаёт максимальное число элементов в списке `FROM`, до которого планировщик будет объединять вложенные запросы с внешним запросом. При меньших значениях сокращается время планирования, но план запроса может стать менее эффективным. По умолчанию это значение равно восьми. За дополнительными сведениями обратитесь к Разделу 14.3.

Если это значение сделать равным geqo_threshold или больше, при таком объединении запросов может включиться планировщик GEQO и в результате будет получен неоптимальный план. См. Подраздел 18.7.3.

`join_collapse_limit` (`integer`)

Задаёт максимальное количество элементов в списке `FROM`, до достижения которого планировщик будет сносить в него явные конструкции `JOIN` (за исключением `FULL JOIN`). При меньших значениях сокращается время планирования, но план запроса может стать менее эффективным.

По умолчанию эта переменная имеет то же значение, что и `from_collapse_limit`, и это приемлемо в большинстве случаев. При значении, равном 1, предложения `JOIN` переставляться не будут, так что явно заданный в запросе порядок соединений определит фактический порядок, в котором будут соединяться отношения. Так как планировщик не всегда выбирает оптимальный порядок соединений, опытные пользователи могут временно задать для этой переменной значение 1, а затем явно определить желаемый порядок. За дополнительными сведениями обратитесь к Разделу 14.3.

Если это значение сделать равным geqo_threshold или больше, при таком объединении запросов может включиться планировщик GEQO и в результате будет получен неоптимальный план. См. Подраздел 18.7.3.

`force_parallel_mode` (`enum`)

Позволяет распараллеливать запрос в целях тестирования, даже когда от этого не ожидается никакого выигрыша в скорости. Допустимые значения параметра `force_parallel_mode` — `off` (использовать параллельный режим только когда ожидается увеличение производительности), `on` (принудительно распараллеливать все запросы, для которых это безопасно) и `regress` (как `on`, но с дополнительными изменениями поведения, описанными ниже).

Говоря точнее, со значением `on` узел `Gather` добавляется в вершину любого плана запроса, для которого допускается распараллеливание, так что запрос выполняется внутри параллельного исполнителя. Даже когда параллельный исполнитель недоступен или не может быть использован, такие операции, как запуск подтранзакции, которые не должны выполняться в контексте параллельного запроса, не будут выполняться в этом режиме, если только

планировщик не решит, что это приведёт к ошибке запроса. Если при включении этого параметра возникают ошибки или выдаются неожиданные результаты, вероятно, некоторые функции, задействованные в этом запросе, нужно пометить как PARALLEL UNSAFE (или, возможно, PARALLEL RESTRICTED).

Значение regress действует так же, как и значение on, с некоторыми дополнительными особенностями, предназначенными для облегчения автоматического регрессионного тестирования. Обычно сообщения от параллельных исполнителей включают строку контекста, отмечающую это, но значение regress подавляет эту строку, так что вывод не отличается от выполнения в не параллельном режиме. Кроме того, узлы Gather, добавляемые в планы с этим значением параметра, скрываются в выводе EXPLAIN, чтобы вывод соответствовал тому, что будет получен при отключении этого параметра (со значением off).

18.8. Регистрация ошибок и протоколирование работы сервера

18.8.1. Куда протоколировать

log_destination (string)

Postgres Pro поддерживает несколько методов протоколирования сообщений сервера: stderr, csvlog и syslog. На Windows также поддерживается eventlog. В качестве значения log_destination указывается один или несколько методов протоколирования, разделённых запятыми. По умолчанию используется stderr. Параметр можно задать только в конфигурационных файлах или в командной строке при запуске сервера.

Если в log_destination включено значение csvlog, то протоколирование ведётся в формате CSV (разделённые запятыми значения). Это удобно для программной обработки журнала. Подробнее об этом в Подразделе 18.8.4. Для вывода в формате CSV должен быть включён logging_collector.

Если присутствует указание stderr или csvlog, создаётся файл current_logfiles, в который записывается расположение файла(ов) журнала, в настоящее время используемого сборщиком сообщений для соответствующего назначения. Это позволяет легко определить, какие файлы журнала используются в данный момент экземпляром сервера. Например, он может иметь такое содержание:

```
stderr log/postgresql.log
csvlog log/postgresql.csv
```

current_logfiles переписывается когда при прокрутке создаётся новый файл журнала или когда изменяется значение log_destination. Он удаляется, когда в log_destination не задаётся ни stderr, ни csvlog, а также когда сборщик сообщений отключён.

> **Примечание**
>
> В большинстве систем Unix потребуется изменить конфигурацию системного демона syslog для использования варианта syslog в log_destination. Для указания типа протоколируемой программы (facility), Postgres Pro может использовать значения с LOCAL0 по LOCAL7 (см. syslog_facility). Однако на большинстве платформ конфигурация syslog по умолчанию не учитывает сообщения подобного типа. Чтобы это работало, потребуется добавить в конфигурацию демона syslog что-то подобное:
>
> ```
> local0.*    /var/log/postgresql
> ```
>
> Для использования eventlog в log_destination на Windows, необходимо зарегистрировать источник событий и его библиотеку в операционной системе. Тогда Windows Event Viewer сможет отображать сообщения журнала событий. Подробнее в Разделе 17.11.

`logging_collector` (`boolean`)

Параметр включает сборщик сообщений (*logging collector*). Это фоновый процесс, который собирает отправленные в stderr сообщения и перенаправляет их в журнальные файлы. Такой подход зачастую более полезен чем запись в syslog, поскольку некоторые сообщения в syslog могут не попасть. (Типичный пример с сообщениями об ошибках динамического связывания, другой пример — ошибки в скриптах типа `archive_command`.) Для установки параметра требуется перезапуск сервера.

> **Примечание**
>
> Можно обойтись без сборщика сообщений и просто писать в stderr. Сообщения будут записываться в место, куда направлен поток stderr. Такой способ подойдёт только для небольших объёмов протоколирования, потому что не предоставляет удобных средств для организации ротации журнальных файлов. Кроме того, на некоторых платформах отказ от использования сборщика сообщений может привести к потере или искажению сообщений, так как несколько процессов, одновременно пишущих в один журнальный файл, могут перезаписывать информацию друг друга.

> **Примечание**
>
> Сборщик спроектирован так, чтобы сообщения никогда не терялись. А это значит, что при очень высокой нагрузке, серверные процессы могут быть заблокированы при попытке отправить сообщения во время сбоя фонового процесса сборщика. В противоположность этому, syslog предпочитает удалять сообщения, при невозможности их записать. Поэтому часть сообщений может быть потеряна, но система не будет блокироваться.

`log_directory` (`string`)

При включённом `logging_collector`, определяет каталог, в котором создаются журнальные файлы. Можно задавать как абсолютный путь, так и относительный от каталога данных кластера. Параметр можно задать только в конфигурационных файлах или в командной строке при запуске сервера. Значение по умолчанию — `log`.

`log_filename` (`string`)

При включённом `logging_collector` задаёт имена журнальных файлов. Значение трактуется как строка формата в функции `strftime`, поэтому в ней можно использовать спецификаторы `%` для включения в имена файлов информации о дате и времени. (При наличии зависящих от часового пояса спецификаторов `%` будет использован пояс, заданный в log_timezone.) Поддерживаемые спецификаторы `%` похожи на те, что перечислены в описании *strftime* спецификации Open Group. Обратите внимание, что системная функция `strftime` напрямую не используется. Поэтому нестандартные, специфичные для платформы особенности не будут работать. Значение по умолчанию `postgresql-%Y-%m-%d_%H%M%S.log`.

Если для задания имени файлов не используются спецификаторы `%`, то для избежания переполнения диска, следует использовать утилиты для ротации журнальных файлов. В версиях до 8.4, при отсутствии спецификаторов `%`, PostgreSQL автоматически добавлял время в формате Epoch к имени файла. Сейчас в этом больше нет необходимости.

Если в `log_destination` включён вывод в формате CSV, то к имени журнального файла будет добавлено расширение `.csv`. (Если `log_filename` заканчивается на `.log`, то это расширение заменится на `.csv`.)

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

log_file_mode (integer)

В системах Unix задаёт права доступа к журнальным файлам, при включённом logging_collector. (В Windows этот параметр игнорируется.) Значение параметра должно быть числовым, в формате команд chmod и umask. (Для восьмеричного формата, требуется задать лидирующий 0 (ноль).)

Права доступа по умолчанию 0600, т. е. только владелец сервера может читать и писать в журнальные файлы. Также, может быть полезным значение 0640, разрешающее чтение файлов членам группы. Однако чтобы установить такое значение, нужно каталог для хранения журнальных файлов (log_directory) вынести за пределы каталога данных кластера. В любом случае нежелательно открывать для всех доступ на чтение журнальных файлов, так как они могут содержать конфиденциальные данные.

Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера.

log_rotation_age (integer)

Определяет максимальное время жизни отдельного журнального файла, при включённом logging_collector. После того как прошло заданное количество минут, создаётся новый журнальный файл. Для запрета создания нового файла по прошествии определённого времени, нужно установить значение 0. Параметр можно задать только в конфигурационных файлах или в командной строке при запуске сервера.

log_rotation_size (integer)

Определяет максимальный размер отдельного журнального файла, при включённом logging_collector. После того как заданное количество килобайт записано в текущий файл, создаётся новый журнальный файл. Для запрета создания нового файла при превышении определённого размера, нужно установить значение 0. Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера.

log_truncate_on_rotation (boolean)

Если параметр logging_collector включён, Postgres Pro будет перезаписывать существующие журнальные файлы, а не дописывать в них. Однако перезапись при переключении на новый файл возможна только в результате ротации по времени, но не при старте сервера или ротации по размеру файла. При выключенном параметре всегда продолжается запись в существующий файл. Например, включение этого параметра в комбинации с log_filename равным postgresql-%H.log, приведёт к генерации 24-х часовых журнальных файлов, которые циклически перезаписываются. Параметр можно задать только в конфигурационных файлах или в командной строке при запуске сервера.

Пример: для хранения журнальных файлов в течение 7 дней, по одному файлу на каждый день с именами вида server_log.Mon, server_log.Tue и т. д., а также с автоматической перезаписью файлов прошлой недели, нужно установить log_filename в server_log.%a, log_truncate_on_rotation в on и log_rotation_age в 1440.

Пример: для хранения журнальных файлов в течение 24 часов, по одному файлу на час, с дополнительной возможностью переключения файла при превышения 1ГБ, установите log_filename в server_log.%H%M, log_truncate_on_rotation в on, log_rotation_age в 60 и log_rotation_size в 1000000. Добавление %M в log_filename позволит при переключении по размеру указать другое имя файла в пределах одного часа.

syslog_facility (enum)

При включённом протоколировании в syslog, этот параметр определяет значение «facility». Допустимые значения LOCAL0, LOCAL1, LOCAL2, LOCAL3, LOCAL4, LOCAL5, LOCAL6, LOCAL7. По умолчанию используется LOCAL0. Подробнее в документации на системный демон syslog. Параметр можно задать только в конфигурационных файлах или в командной строке при запуске сервера.

`syslog_ident` (`string`)

При включённом протоколировании в syslog, этот параметр задаёт имя программы, которое будет использоваться в syslog для идентификации сообщений относящихся к Postgres Pro. По умолчанию используется `postgres`. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`syslog_sequence_numbers` (`boolean`)

Когда сообщения выводятся в syslog и этот параметр включён (по умолчанию), все сообщения будут предваряться последовательно увеличивающимися номерами (например, `[2]`). Это позволяет обойти подавление повторов «--- последнее сообщение повторилось N раз ---», которое по умолчанию осуществляется во многих реализациях syslog. В более современных реализациях syslog подавление повторных сообщений можно настроить (например, в rsyslog есть директива `$RepeatedMsgReduction`), так что это может излишне. Если же вы действительно хотите, чтобы повторные сообщения подавлялись, вы можете отключить этот параметр.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`syslog_split_messages` (`boolean`)

Когда активен вывод сообщений в syslog, этот параметр определяет, как будут доставляться сообщения. Если он включён (по умолчанию), сообщения разделяются по строкам, а длинные строки разбиваются на строки не длиннее 1024 байт, что составляет типичное ограничение размера для традиционных реализаций syslog. Когда он отключён, сообщения сервера Postgres Pro передаются службе syslog как есть, и она должна сама корректно воспринять потенциально длинные сообщения.

Если syslog в итоге выводит сообщения в текстовый файл, результат будет тем же и лучше оставить этот параметр включённым, так как многие реализации syslog не способны обрабатывать большие сообщения или их нужно специально настраивать для этого. Но если syslog направляет сообщения в некоторую другую среду, может потребоваться или будет удобнее сохранять логическую целостность сообщений.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`event_source` (`string`)

При включённом протоколировании в event log, этот параметр задаёт имя программы, которое будет использоваться в журнале событий для идентификации сообщений относящихся к Postgres Pro. По умолчанию используется `Postgres Pro`. Параметр можно задать только в конфигурационных файлах или в командной строке при запуске сервера.

18.8.2. Когда протоколировать

`log_min_messages` (`enum`)

Управляет минимальным уровнем сообщений, записываемых в журнал сервера. Допустимые значения `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL` и `PANIC`. Каждый из перечисленных уровней включает все идущие после него. Чем дальше в этом списке уровень сообщения, тем меньше сообщений будет записано в журнал сервера. По умолчанию используется `WARNING`. Обратите внимание, позиция `LOG` здесь отличается от принятой в client_min_messages. Только суперпользователи могут изменить этот параметр.

`log_min_error_statement` (`enum`)

Управляет тем, какие SQL-операторы, завершившиеся ошибкой, записываются в журнал сервера. SQL-оператор будет записан в журнал, если он завершится ошибкой с указанным уровнем важности или выше. Допустимые значения: `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL` и `PANIC`. По умолчанию используется `ERROR`. Это означает, что в журнал сервера будут записаны все операторы, завершившиеся сообщением с

уровнем важности ERROR, LOG, FATAL и PANIC. Чтобы фактически отключить запись операторов с ошибками, установите для этого параметра значение PANIC. Изменить этот параметр могут только суперпользователи.

log_min_duration_statement (integer)

Записывает в журнал продолжительность выполнения всех команд, время работы которых равно или превышает указанное количество миллисекунд. Значение 0 (ноль) заставляет записывать продолжительность работы всех команд. Значение -1 (по умолчанию) запрещает регистрировать продолжительность выполнения операторов. Например, при значении 250ms, все команды, которые выполняются за 250 миллисекунд и дольше будут записаны в журнал сервера. Включение параметра полезно для выявления плохо оптимизированных запросов в приложении. Только суперпользователи могут изменить этот параметр.

Для клиентов, использующих расширенный протокол запросов, будет записываться продолжительность фаз: разбор, связывание и выполнение.

> **Примечание**
>
> При использовании совместно с log_statement, текст SQL-операторов будет записываться только один раз (от использования log_statement) и не будет задублирован в сообщении о длительности выполнения. Если не используется вывод в syslog, то рекомендуется в log_line_prefix включить идентификатор процесса или сессии. Это позволит связать текст запроса с записью о продолжительности выполнения, которая появится позже.

В Таблице 18.2 поясняются уровни важности сообщений в Postgres Pro. Также в этой таблице показано, как эти уровни транслируются в системные при использовании syslog или eventlog в Windows.

Таблица 18.2. Уровни важности сообщений

Уровень	Использование	syslog	eventlog
DEBUG1..DEBUG5	Более детальная информация для разработчиков. Чем больше номер, тем детальнее.	DEBUG	INFORMATION
INFO	Неявно запрошенная пользователем информация, например вывод команды VACUUM VERBOSE.	INFO	INFORMATION
NOTICE	Информация, которая может быть полезной пользователям. Например, уведомления об усечении длинных идентификаторов.	NOTICE	INFORMATION
WARNING	Предупреждения о возможных проблемах. Например, COMMIT вне транзакционного блока.	NOTICE	WARNING
ERROR	Сообщает об ошибке, из-за которой прервана текущая команда.	WARNING	ERROR

Уровень	Использование	syslog	eventlog
LOG	Информация, полезная для администраторов. Например, выполнение контрольных точек.	INFO	INFORMATION
FATAL	Сообщает об ошибке, из-за которой прервана текущая сессия.	ERR	ERROR
PANIC	Сообщает об ошибке, из-за которой прерваны все сессии.	CRIT	ERROR

18.8.3. Что протоколировать

> **Примечание**
>
> Выбирая, что будет записываться в протокол, важно учитывать возможные риски безопасности; подробнее об этом говорится в Разделе 23.3.

application_name (string)

application_name — это любая строка длиной не более NAMEDATALEN символов (64 символа при стандартной сборке). Обычно устанавливается приложением при подключении к серверу. Значение отображается в представлении pg_stat_activity и добавляется в журнал сервера, при использовании формата CSV. Для прочих форматов, application_name можно добавить в журнал через параметр log_line_prefix. Значение application_name может содержать только печатные ASCII символы. Остальные символы будут заменены знаками вопроса (?).

debug_print_parse (boolean)
debug_print_rewritten (boolean)
debug_print_plan (boolean)

Эти параметры включают вывод различной отладочной информации. А именно: вывод дерева запроса, дерево запроса после применения правил или плана выполнения запроса, соответственно. Все эти сообщения имеют уровень LOG. Поэтому, по умолчанию, они записываются в журнал сервера, но не отправляются клиенту. Отправку клиенту можно настроить через client_min_messages и/или log_min_messages. По умолчанию параметры выключены.

debug_pretty_print (boolean)

Включает выравнивание сообщений, выводимых debug_print_parse, debug_print_rewritten или debug_print_plan. В результате сообщения легче читать, но они значительно длиннее, чем в формате «compact», который используется при выключенном значении. По умолчанию включён.

log_checkpoints (boolean)

Включает протоколирование выполнения контрольных точек и точек перезапуска сервера. При этом записывается некоторая статистическая информация. Например, число записанных буферов и время, затраченное на их запись. Параметр можно задать только в конфигурационных файлах или в командной строке при запуске сервера. По умолчанию выключен.

log_connections (boolean)

Включает протоколирование всех попыток подключения к серверу, в том числе успешного завершения аутентификации клиентов. Изменить его можно только в начале сеанса и сделать это могут только суперпользователи. Значение по умолчанию — off.

> **Примечание**
>
> Некоторые программы, например psql, предпринимают две попытки подключения (первая для определения, нужен ли пароль). Поэтому дублирование сообщения «connection received» не обязательно говорит о наличии проблемы.

`log_disconnections` (`boolean`)

Включает протоколирование завершения сеанса. В журнал выводится примерно та же информация, что и с `log_connections`, плюс длительность сеанса. Изменить этот параметр можно только в начале сеанса и сделать это могут только суперпользователи. Значение по умолчанию — `off`.

`log_duration` (`boolean`)

Записывает продолжительность каждой завершённой команды. По умолчанию выключен. Только суперпользователи могут изменить этот параметр.

Для клиентов, использующих расширенный протокол запросов, будет записываться продолжительность фаз: разбор, связывание и выполнение.

> **Примечание**
>
> Включение этого параметра не равнозначно установке нулевого значения для log_min_duration_statement. Разница в том, что при превышении значения `log_min_duration_statement` в журнал записывается текст запроса, а при включении данного параметра — нет. Таким образом при `log_duration = on` и положительном значении `log_min_duration_statement` в журнал записывается длительность всех команд, а текст запросов — только для команд с длительностью, превысившей предел. Такое поведение может оказаться полезным при сборе статистики в условиях большой нагрузки.

`log_error_verbosity` (`enum`)

Управляет количеством детальной информации, записываемой в журнал сервера для каждого сообщения. Допустимые значения: `TERSE`, `DEFAULT` и `VERBOSE`. Каждое последующее значение добавляет больше полей в выводимое сообщение. Для `TERSE` из сообщения об ошибке исключаются поля `DETAIL`, `HINT`, `QUERY` и `CONTEXT`. Для `VERBOSE` в сообщение включается код ошибки `SQLSTATE` (см. Приложение А), а также имя файла с исходным кодом, имя функции и номер строки сгенерировавшей ошибку. Только суперпользователи могут изменить этот параметр.

`log_hostname` (`boolean`)

По умолчанию, сообщения журнала содержат лишь IP-адрес подключившегося клиента. При включении этого параметра, дополнительно будет фиксироваться и имя сервера. Обратите внимание, что в зависимости от применяемого способа разрешения имён, это может отрицательно сказаться на производительности. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`log_line_prefix` (`string`)

Строка, в стиле функции `printf`, которая выводится в начале каждой строки журнала сообщений. С символов `%` начинаются управляющие последовательности, которые заменяются статусной информацией, описанной ниже. Неизвестные управляющие последовательности игнорируются. Все остальные символы напрямую копируются в выводимую строку. Некоторые управляющие последовательности используются только для пользовательских процессов и будут игнорироваться фоновыми процессами, например основным процессом сервера.

Статусная информация может быть выровнена по ширине влево или вправо указанием числа после % и перед кодом последовательности. Отрицательное число дополняет значение пробелами справа до заданной ширины, а положительное число — слева. Выравнивание может быть полезно для улучшения читаемости. Этот параметр можно задать только в файле `postgresql.conf` или в командной строке при запуске сервера. Со значением по умолчанию, `'%m [%p] '`, в журнал выводится метка времени и идентификатор процесса.

Спецсимвол	**Назначение**	**Только для пользовательского процесса**
`%a`	Имя приложения (application_name)	да
`%u`	Имя пользователя	да
`%d`	Имя базы данных	да
`%r`	Имя удалённого узла или IP-адрес, а также номер порта	да
`%h`	Имя удалённого узла или IP-адрес	да
`%p`	Идентификатор процесса	нет
`%t`	Штамп времени, без миллисекунд	нет
`%m`	Штамп времени, с миллисекундами	нет
`%n`	Штамп времени, с миллисекундами (в виде времени Unix)	нет
`%i`	Тег команды: тип текущей команды в сессии	да
`%e`	Код ошибки SQLSTATE	нет
`%c`	Идентификатор сессии. Подробности ниже	нет
`%l`	Номер строки журнала для каждой сессии или процесса. Начинается с 1	нет
`%s`	Штамп времени начала процесса	нет
`%v`	Идентификатор виртуальной транзакции (backendID/localXID)	нет
`%x`	Идентификатор транзакции (0 если не присвоен)	нет
`%q`	Ничего не выводит. Непользовательские процессы останавливаются в этой точке. Игнорируется пользовательскими процессами	нет
`%%`	Выводит `%`	нет

`%c` выводит псевдоуникальный номер сеанса, состоящий из двух 4-байтных шестнадцатеричных чисел (без ведущих нулей), разделённых точкой. Эти числа представляют время старта

процесса и идентификатор процесса, поэтому `%c` можно использовать для экономии места при выводе этих значений. Например, для получения идентификатора сеанса из `pg_stat_activity` используйте запрос:

```
SELECT to_hex(trunc(EXTRACT(EPOCH FROM backend_start))::integer) || '.' ||
       to_hex(pid)
FROM pg_stat_activity;
```

> **Подсказка**
>
> Последним символом в `log_line_prefix` лучше оставлять пробел, чтобы отделить от остальной строки. Можно использовать и символы пунктуации.

> **Подсказка**
>
> Syslog также формирует штамп времени и идентификатор процесса, поэтому вероятно нет смысла использовать соответствующие управляющие последовательности при использовании syslog.

> **Подсказка**
>
> Спецпоследовательность `%q` полезна для включения информации, которая существует только в контексте сеанса (обслуживающего процесса), например, имя пользователя или базы данных. Например:
>
> ```
> log_line_prefix = '%m [%p] %q%u@%d/%a '
> ```

`log_lock_waits` (`boolean`)

Определяет, нужно ли фиксировать в журнале события, когда сеанс ожидает получения блокировки дольше, чем указано в deadlock_timeout. Это позволяет выяснить, не связана ли низкая производительность с ожиданием блокировок. По умолчанию отключено. Только суперпользователи могут изменить этот параметр.

`log_statement` (`enum`)

Управляет тем, какие SQL-команды записывать в журнал. Допустимые значения: `none` (отключено), `ddl`, `mod` и `all` (все команды). `ddl` записывает все команды определения данных, такие как `CREATE`, `ALTER`, `DROP`. `mod` записывает все команды `ddl`, а также команды изменяющие данные, такие как `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` и `COPY FROM`. `PREPARE`, `EXECUTE` и `EXPLAIN ANALYZE` также записываются, если вызваны для команды соответствующего типа. Если клиент использует расширенный протокол запросов, то запись происходит на фазе выполнения и содержит значения всех связанных переменных (если есть символы одиночных кавычек, то они дублируются).

По умолчанию `none`. Только суперпользователи могут изменить этот параметр.

> **Примечание**
>
> Команды с синтаксическими ошибками не записываются, даже если `log_statement = all`, так как сообщение формируется только после выполнения предварительного разбора, определяющего тип команды. При расширенном протоколе запросов, похожим образом не будут записываться команды, неуспешно завершившиеся до фазы выполнения (например, при разборе или построении плана запроса). Для включения в журнал таких команд установите `log_min_error_statement` в `ERROR` (или ниже).

Записываемые в журнал команды могут раскрывать конфиденциальные данные и даже содержать незашифрованные пароли.

`log_replication_commands` (`boolean`)

Включает запись в журнал сервера всех команд репликации. Подробнее о командах репликации можно узнать в Разделе 50.4. Значение по умолчанию — `off`. Изменить этот параметр могут только суперпользователи.

`log_temp_files` (`integer`)

Управляет включением в журнал информации об именах и размерах временных файлов. Временные файлы могут использоваться для сортировок, хеширования и временного хранения результатов запросов. На каждый временный файл, при его удалении, в журнал записывается отдельное сообщение. Значение 0 говорит о том, что нужно записывать информацию о всех временных файлах. Положительное значение задаёт размер временных файлов в килобайтах, при достижении или превышении которого, информация о временном файле будет записана. Значение по умолчанию -1, что отключает запись информации о временных файлах. Только суперпользователи могут изменить этот параметр.

`log_timezone` (`string`)

Устанавливает часовой пояс для штампов времени при записи в журнал сервера. В отличие от TimeZone, это значение одинаково для всех баз данных кластера, поэтому для всех сессий используются согласованные значения штампов времени. Встроенное значение по умолчанию `GMT`, но оно переопределяется в `postgresql.conf`: initdb записывает в него значение, соответствующее системной среде. Подробнее об этом в Подразделе 8.5.3. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

18.8.4. Использование вывода журнала в формате CSV

Добавление `csvlog` в `log_destination` делает удобным загрузку журнальных файлов в таблицу базы данных. Строки журнала представляют собой значения разделённые запятыми (CSV формат) со следующими полями: штамп времени с миллисекундами; имя пользователя; имя базы данных; идентификатор процесса; клиентский узел:номер порта; идентификатор сессии; номер строки каждой сессии; тег команды; время начала сессии; виртуальный идентификатор транзакции; идентификатор транзакции; уровень важности ошибки; код ошибки SQLSTATE; сообщение об ошибке; подробности к сообщению об ошибке; подсказка к сообщению об ошибке; внутренний запрос, приведший к ошибке (если есть); номер символа внутреннего запроса, где произошла ошибка; контекст ошибки; запрос пользователя, приведший к ошибке (если есть и включён `log_min_error_statement`); номер символа в запросе пользователя, где произошла ошибка; расположение ошибки в исходном коде Postgres Pro (если `log_error_verbosity` установлен в `verbose`) и имя приложения. Вот пример определения таблицы, для хранения журналов в формате CSV:

```
CREATE TABLE postgres_log
(
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
```

```
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text,
  PRIMARY KEY (session_id, session_line_num)
);
```

Для загрузки журнального файла в такую таблицу можно использовать команду `COPY FROM`:

```
COPY postgres_log FROM '/full/path/to/logfile.csv' WITH csv;
```

Также можно обратиться к этому файлу как к сторонней таблице, используя поставляемый модуль file_fdw.

Для упрощения загрузки журналов в CSV формате используйте следующее:

1. Установите для `log_filename` и `log_rotation_age` значения, гарантирующие согласованную и предсказуемую схему именования журнальных файлов. Зная, какие имена будут у журнальных файлов, можно определить, когда конкретный файл заполнен и готов к загрузке.
2. Установите `log_rotation_size` в 0, чтобы запретить ротацию файлов по достижении определённого размера, так как это делает непредсказуемой схему именования журнальных файлов.
3. Установите `log_truncate_on_rotation` в `on`, чтобы новые сообщения не смешивались со старыми при переключении на существующий файл.
4. Определение таблицы содержит первичный ключ. Это полезно для предотвращения случайной повторной загрузки данных. Команда `COPY` фиксирует изменения один раз, поэтому любая ошибка приведёт к отмене всей загрузки. Если сначала загрузить неполный журнальный файл, то его повторная загрузка (по заполнении) приведёт к нарушению первичного ключа и, следовательно, к ошибке загрузки. Поэтому необходимо дожидаться окончания записи в журнальный файл перед началом загрузки. Похожим образом предотвращается случайная загрузка частично сформированной строки сообщения, что также приведёт к сбою в команде `COPY`.

18.8.5. Заголовок процесса

Эти параметры влияют на то, как формируются заголовки серверных процессов. Заголовок процесса обычно можно наблюдать в программах типа ps, а в Windows — в Process Explorer. За подробностями обратитесь к Разделу 27.1.

`cluster_name` (`string`)

Задаёт имя кластера, которое отображается в заголовке процесса для всех серверных процессов данного кластера. Это имя может быть любой строкой не длиннее `NAMEDATALEN` символов (64 символа в стандартной сборке). В строке `cluster_name` могут использоваться только печатаемые ASCII-символы. Любой другой символ будет заменён символом вопроса (`?`). Если этот параметр содержит пустую строку `''` (это значение по умолчанию), никакое имя не выводится. Этот параметр можно задать только при запуске сервера.

`update_process_title` (`boolean`)

Включает изменение заголовка процесса при получении сервером каждой очередной команды SQL. На большинстве платформ он включён (имеет значение `on`), но в Windows по умолчанию

выключен (off) ввиду больших издержек на изменение этого заголовка. Изменить этот параметр могут только суперпользователи.

18.9. Статистика времени выполнения

18.9.1. Сбор статистики по запросам и индексам

Эти параметры управляет функциями сбора статистики на уровне сервера. Когда ведётся сбор статистики, собираемые данные можно просмотреть в семействе системных представлений pg_stat и pg_statio. За дополнительными сведениями обратитесь к Главе 27.

track_activities (boolean)

Включает сбор сведений о текущих командах, выполняющихся во всех сеансах (в частности, отслеживается время запуска команды). По умолчанию этот параметр включён. Заметьте, что даже когда сбор ведётся, собранная информация доступна не для всех пользователей, а только для суперпользователей и пользователей, включённых в роль pg_read_all_stats. Также пользователям доступна информация о командах в их сеансах (включая сеансы ролей, в которые эти пользователи включены). Поэтому это не должно повлечь риски, связанные с безопасностью. Изменить этот параметр могут только суперпользователи.

track_activity_query_size (integer)

Задаёт число байт, которое будет зарезервировано для отслеживания выполняемой в данной момент команды в каждом активном сеансе, для поля pg_stat_activity.query. Значение по умолчанию — 1024. Задать этот параметр можно только при запуске сервера.

track_counts (boolean)

Включает сбор статистики активности в базе данных. Этот параметр по умолчанию включён, так как собранная информация требуется демону автоочистки. Изменить этот параметр могут только суперпользователи.

track_io_timing (boolean)

Включает замер времени операций ввода/вывода. Этот параметр по умолчанию отключён, так как для этого требуется постоянно запрашивать текущее время у операционной системы, что может значительно замедлить работу на некоторых платформах. Для оценивания издержек замера времени на вашей платформе можно воспользоваться утилитой pg_test_timing. Статистику ввода/вывода можно получить через представление pg_stat_database, в выводе EXPLAIN (когда используется параметр BUFFERS) и через представление pg_stat_statements. Изменить этот параметр могут только суперпользователи.

track_functions (enum)

Включает подсчёт вызовов функций и времени их выполнения. Значение pl включает отслеживание только функций на процедурном языке, а all — также функций на языках SQL и C. Значение по умолчанию — none, то есть сбор статистики по функциям отключён. Изменить этот параметр могут только суперпользователи.

> **Примечание**
>
> Функции на языке SQL, достаточно простые для «внедрения» в вызывающий запрос, отслеживаться не будут вне зависимости от этого параметра.

stats_temp_directory (string)

Задаёт каталог, в котором будут храниться временные данные статистики. Этот путь может быть абсолютным или задаваться относительно каталога данных. Значение по умолчанию —

pg_stat_tmp. Если разместить целевой каталог в файловой системе в ОЗУ, это снизит нагрузку на физическое дисковое хранилище и может увеличить быстродействие. Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера.

18.9.2. Мониторинг статистики

log_statement_stats (boolean)
log_parser_stats (boolean)
log_planner_stats (boolean)
log_executor_stats (boolean)

Эти параметры включают вывод статистики по производительности соответствующего модуля в протокол работы сервера. Это грубый инструмент профилирования, похожий на функцию getrusage() в операционной системе. Параметр log_statement_stats включает вывод общей статистики по операторам, тогда как другие управляют статистикой по модулям (разбор, планирование, выполнение). Включить log_statement_stats одновременно с параметрами, управляющими модулями, нельзя. По умолчанию все эти параметры отключены. Изменить эти параметры могут только суперпользователи.

18.10. Автоматическая очистка

Эти параметры управляют поведением механизма *автоочистки*. За дополнительными сведениями обратитесь к Подразделу 23.1.6. Заметьте, что многие из этих параметров могут быть переопределены на уровне таблиц; см. Подраздел «Параметры хранения».

autovacuum (boolean)

Управляет состоянием демона, запускающего автоочистку. По умолчанию он включён, но чтобы автоочистка работала, нужно также включить track_counts. Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера. Однако автоочистку можно отключить для отдельных таблиц, изменив их параметры хранения.

Заметьте, что даже если этот параметр отключён, система будет запускать процессы автоочистки, когда это необходимо для предотвращения зацикливания идентификаторов транзакций. За дополнительными сведениями обратитесь к Подразделу 23.1.5.

log_autovacuum_min_duration (integer)

Задаёт время выполнения действия автоочистки (в миллисекундах), при превышении которого информация об этом действии записывается в протокол. При нулевом значении в протоколе фиксируются все действия автоочистки. Значение -1 (по умолчанию) отключает протоколирование действий автоочистки. Например, если задать значение 250ms, в протоколе будут фиксироваться все операции автоматической очистки и анализа, выполняемые дольше 250 мс. Кроме того, когда этот параметр имеет любое значение, отличное от -1, в протокол будет записываться сообщение в случае пропуска действия автоочистки из-за конфликтующей блокировки. Таким образом, включение этого параметра позволяет отслеживать активность автоочистки. Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера. Однако его можно переопределить для отдельных таблиц, изменив их параметры хранения.

autovacuum_max_workers (integer)

Задаёт максимальное число процессов автоочистки (не считая процесс, запускающий автоочистку), которые могут выполняться одновременно. По умолчанию это число равно трём. Задать этот параметр можно только при запуске сервера.

autovacuum_naptime (integer)

Задаёт минимальную задержку между двумя запусками автоочистки для отдельной базы данных. Демон автоочистки проверяет базу данных через заданный интервал времени и выдаёт команды VACUUM и ANALYZE, когда это требуется для таблиц этой базы. Задержка задаётся в

секундах и по умолчанию равна одной минуте (1min). Этот параметр можно задать только в postgresql.conf или в командной строке при запуске сервера.

autovacuum_vacuum_threshold (integer)

Задаёт минимальное число изменённых или удалённых кортежей, при котором будет выполняться VACUUM для отдельно взятой таблицы. Значение по умолчанию — 50 кортежей. Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера. Однако данное значение можно переопределить для избранных таблиц, изменив их параметры хранения.

autovacuum_analyze_threshold (integer)

Задаёт минимальное число добавленных, изменённых или удалённых кортежей, при котором будет выполняться ANALYZE для отдельно взятой таблицы. Значение по умолчанию — 50 кортежей. Этот параметр можно задать только в postgresql.conf или в командной строке при запуске сервера. Однако данное значение можно переопределить для избранных таблиц, изменив их параметры хранения.

autovacuum_vacuum_scale_factor (floating point)

Задаёт процент от размера таблицы, который будет добавляться к autovacuum_vacuum_threshold при выборе порога срабатывания команды VACUUM. Значение по умолчанию — 0.2 (20% от размера таблицы). Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера. Однако данное значение можно переопределить для избранных таблиц, изменив их параметры хранения.

autovacuum_analyze_scale_factor (floating point)

Задаёт процент от размера таблицы, который будет добавляться к autovacuum_analyze_threshold при выборе порога срабатывания команды ANALYZE. Значение по умолчанию — 0.1 (10% от размера таблицы). Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера. Однако данное значение можно переопределить для избранных таблиц, изменив их параметры хранения.

autovacuum_freeze_max_age (integer)

Задаёт максимальный возраст (в транзакциях) для поля pg_class.relfrozenxid некоторой таблицы, при достижении которого будет запущена операция VACUUM для предотвращения зацикливания идентификаторов транзакций в этой таблице. Заметьте, что система запустит процессы автоочистки для предотвращения зацикливания, даже если для всех других целей автоочистка отключена.

При очистке могут также удаляться старые файлы из подкаталога pg_xact, поэтому значение по умолчанию сравнительно мало — 200 миллионов транзакций. Задать этот параметр можно только при запуске сервера, но для отдельных таблиц его можно определить по-другому, изменив их параметры хранения. За дополнительными сведениями обратитесь к Подразделу 23.1.5.

autovacuum_multixact_freeze_max_age (integer)

Задаёт максимальный возраст (в мультитранзакциях) для поля pg_class.relminmxid таблицы, при достижении которого будет запущена операция VACUUM для предотвращения зацикливания идентификаторов мультитранзакций в этой таблице. Заметьте, что система запустит процессы автоочистки для предотвращения зацикливания, даже если для всех других целей автоочистка отключена.

При очистке мультитранзакций могут также удаляться старые файлы из подкаталогов pg_multixact/members и pg_multixact/offsets, поэтому значение по умолчанию сравнительно мало — 400 миллионов мультитранзакций. Этот параметр можно задать только при запуске сервера, но для отдельных таблиц его можно определить по-другому, изменив их параметры хранения. За дополнительными сведениями обратитесь к Подразделу 23.1.5.1.

`autovacuum_vacuum_cost_delay` (`integer`)

Задаёт задержку при превышении предела стоимости, которая будет применяться при автоматических операциях `VACUUM`. При значении -1 применяется обычная задержка vacuum_cost_delay. Значение по умолчанию — 20 миллисекунд. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера. Однако его можно переопределить для отдельных таблиц, изменив их параметры хранения.

`autovacuum_vacuum_cost_limit` (`integer`)

Задаёт предел стоимости, который будет учитываться при автоматических операциях `VACUUM`. При значении -1 (по умолчанию) применяется обычное значение vacuum_cost_limit. Заметьте, что это значение распределяется пропорционально среди всех работающих процессов автоочистки, если их больше одного, так что сумма ограничений всех процессов никогда не превосходит данный предел. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера. Однако его можно переопределить для отдельных таблиц, изменив их параметры хранения.

18.11. Параметры клиентских сеансов по умолчанию

18.11.1. Поведение команд

`client_min_messages` (`enum`)

Управляет минимальным уровнем сообщений, посылаемых клиенту. Допустимые значения `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `LOG`, `NOTICE`, `WARNING` и `ERROR`. Каждый из перечисленных уровней включает все идущие после него. Чем дальше в этом списке уровень сообщения, тем меньше сообщений будет посылаться клиенту. По умолчанию используется `NOTICE`. Обратите внимание, позиция `LOG` здесь отличается от принятой в log_min_messages.

Сообщения уровня `INFO` передаются клиенту всегда.

`search_path` (`string`)

Эта переменная определяет порядок, в котором будут просматриваться схемы при поиске объекта (таблицы, типа данных, функции и т. д.), к которому обращаются просто по имени, без указания схемы. Если объекты с одинаковым именем находятся в нескольких схемах, использоваться будет тот, что встретится первым при просмотре пути поиска. К объекту, который не относится к схемам, перечисленным в пути поиска, можно обратиться только по полному имени (с точкой), с указанием содержащей его схемы.

Значением `search_path` должен быть список имён схем через запятую. Если для имени, указанного в этом списке, не находится существующая схема, либо пользователь не имеет права `USAGE` для схемы с этим именем, такое имя просто игнорируется.

Если список содержит специальный элемент `$user`, вместо него подставляется схема с именем, возвращаемым функцией `CURRENT_USER`, если такая схема существует и пользователь имеет право `USAGE` для неё. (В противном случае элемент `$user` игнорируется.)

Схема системных каталогов, `pg_catalog`, просматривается всегда, независимо от того, указана она в пути или нет. Если она указана в пути, она просматривается в заданном порядке. Если же `pg_catalog` отсутствует в пути, эта схема будет просматриваться *перед* остальными элементами пути.

Аналогично всегда просматривается схема временных таблиц текущего сеанса, `pg_temp_`*`nnn`*, если она существует. Её можно включить в путь поиска, указав её псевдоним `pg_temp`. Если она отсутствует в пути, она будет просматриваться первой (даже перед `pg_catalog`). Временная схема просматривается только при поиске отношений (таблиц, представлений, последовательностей и т. д.) и типов данных, но никогда при поиске функций и операторов.

Когда объекты создаются без указания определённой целевой схемы, они помещаются в первую пригодную схему, указанную в search_path. Если путь поиска схем пуст, выдаётся ошибка.

По умолчанию этот параметр имеет значение "$user", public. При таком значении поддерживается совместное использование базы данных (когда пользователи не имеют личных схем, все используют схему public), использование личных схем, а также комбинация обоих вариантов. Другие подходы можно реализовать, изменяя значение пути по умолчанию, либо глобально, либо индивидуально для каждого пользователя.

Более подробно обработка схем описана в Разделе 5.8. В частности, стандартная конфигурация схем подходит только для баз данных с одним пользователем или с взаимно доверяющими пользователями.

Текущее действующее значение пути поиска можно получить, воспользовавшись SQL-функцией current_schemas (см. Раздел 9.25). Это значение может отличаться от значения search_path, так как current_schemas показывает, как были преобразованы элементы, фигурирующие в search_path.

row_security (boolean)

Эта переменная определяет, должна ли выдаваться ошибка при применении политик защиты строк. Со значением on политики применяются в обычном режиме. Со значением off запросы, ограничиваемые минимум одной политикой, будут выдавать ошибку. Значение по умолчанию — on. Значение off рекомендуется, когда ограничение видимости строк чревато некорректными результатами; например, pg_dump устанавливает это значение. Эта переменная не влияет на роли, которые обходят все политики защиты строк, а именно, на суперпользователей и роли с атрибутом BYPASSRLS.

Подробнее о политиках защиты строк можно узнать в описании CREATE POLICY.

default_tablespace (string)

Эта переменная устанавливает табличное пространство по умолчанию, в котором будут создаваться объекты (таблицы и индексы), когда в команде CREATE табличное пространство не указывается явно.

Её значением может быть либо имя табличного пространства, либо пустая строка, подразумевающая использование табличного пространства по умолчанию в текущей базе данных. Если табличное пространство с заданным именем не существует, Postgres Pro будет автоматически использовать табличное пространство по умолчанию. Если используется не пространство по умолчанию, пользователь должен иметь право CREATE для него, иначе он не сможет создавать объекты.

Эта переменная не используется для временных таблиц; для них задействуется temp_tablespaces.

Эта переменная также не используется при создании баз данных. По умолчанию, новая база данных наследует выбор табличного пространства от базы-шаблона, из которой она копируется.

За дополнительными сведениями о табличных пространствах обратитесь к Разделу 21.6.

temp_tablespaces (string)

Эта переменная задаёт табличные пространства, в которых будут создаваться временные объекты (временные таблицы и индексы временных таблиц), когда в команде CREATE табличное пространство не указывается явно. В этих табличных пространствах также создаются временные файлы для внутреннего использования, например, для сортировки больших наборов данных.

Её значение содержит список имён табличных пространств. Когда этот список содержит больше одного имени, Postgres Pro выбирает из этого списка случайный элемент при создании каждого временного объекта; однако при создании последующих объектов внутри транзакции табличные пространства перебираются последовательно. Если в этом списке оказывается пустая строка, Postgres Pro будет автоматически использовать вместо этого элемента табличное пространство по умолчанию для текущей базы данных.

Когда `temp_tablespaces` задаётся интерактивно, указание несуществующего табличного пространства считается ошибкой, как и указание табличного пространства, для которого пользователь не имеет права `CREATE`. Однако при использовании значения, заданного ранее, несуществующие табличные пространства и пространства, для которых у пользователя нет права `CREATE`, просто игнорируются. В частности, это касается значения, заданного в `postgresql.conf`.

По умолчанию значение этой переменной — пустая строка. С таким значением все временные объекты создаются в табличном пространстве по умолчанию, установленном для текущей базы данных.

См. также default_tablespace.

`check_function_bodies` (`boolean`)

Этот параметр обычно включён. Выключение этого параметра (присваивание ему значения `off`) отключает проверку строки с телом функции, передаваемой команде CREATE FUNCTION. Отключение проверки позволяет избежать побочных эффектов процесса проверки и исключить ложные срабатывания из-за таких проблем, как ссылки вперёд. Этому параметру нужно присваивать значение `off` перед загрузкой функций от лица других пользователей; pg_dump делает это автоматически.

`default_transaction_isolation` (`enum`)

Для каждой транзакции в SQL устанавливается уровень изоляции: «read uncommitted», «read committed», «repeatable read» или «serializable». Этот параметр задаёт уровень изоляции, который будет устанавливаться по умолчанию для новых транзакций. Значение этого параметра по умолчанию — «read committed».

Дополнительную информацию вы можете найти в Главе 13 и SET TRANSACTION.

`default_transaction_read_only` (`boolean`)

SQL-транзакции в режиме «только чтение» не могут модифицировать не временные таблицы. Этот параметр определяет, будут ли новые транзакции по умолчанию иметь характеристику «только чтение». Значение этого параметра по умолчанию — `off` (допускается чтение и запись).

За дополнительной информацией обратитесь к SET TRANSACTION.

`default_transaction_deferrable` (`boolean`)

Транзакция, работающая на уровне изоляции `serializable`, в режиме «только чтение» может быть задержана, прежде чем будет разрешено её выполнение. Однако когда она начинает выполняться, для обеспечения сериализуемости не требуется никаких дополнительных усилий, так что коду сериализации ни при каких условиях не придётся прерывать её из-за параллельных изменений, поэтому это вполне подходит для длительных транзакций в режиме «только чтение».

Этот параметр определяет, будет ли каждая новая транзакция по умолчанию откладываемой. В настоящее время его действие не распространяется на транзакции, для которых устанавливается режим «чтение/запись» или уровень изоляции ниже `serializable`. Значение по умолчанию — `off` (выкл.).

За дополнительной информацией обратитесь к SET TRANSACTION.

`transaction_isolation` (`enum`)

Данный параметр отражает уровень изоляции текущей транзакции. В начале каждой транзакции ему присваивается текущее значение default_transaction_isolation. Последующая попытка изменить значение этого параметра равнозначна команде SET TRANSACTION.

`transaction_read_only` (`boolean`)

Данный параметр отражает характеристику «только чтение» для текущей транзакции. В начале каждой транзакции ему присваивается текущее значение default_transaction_read_only. Последующая попытка изменить значение этого параметра равнозначна команде SET TRANSACTION.

`transaction_deferrable` (`boolean`)

Данный параметр отражает характеристику «откладываемости» для текущей транзакции. В начале каждой транзакции ему присваивается текущее значение default_transaction_deferrable. Последующая попытка изменить значение этого параметра равнозначна команде SET TRANSACTION.

`session_replication_role` (`enum`)

Управляет срабатыванием правил и триггеров, связанных с репликацией, в текущем сеансе. Изменение этой переменной требует наличия прав суперпользователя и приводит к сбросу всех ранее кешированных планов запросов. Она может принимать следующие значения `origin` (значение по умолчанию), `replica` и `local`. За дополнительными сведениями обратитесь к ALTER TABLE.

`statement_timeout` (`integer`)

Задаёт максимальное время выполнения оператора (в миллисекундах), начиная с момента получения сервером команды от клиента, по истечении которого оператор прерывается. Если `log_min_error_statement` имеет значение `ERROR` или ниже, оператор, прерванный по тайм-ауту, будет также записан в журнал. При значении, равном нулю (по умолчанию), этот контроль длительности отключается.

Устанавливать значение `statement_timeout` в `postgresql.conf` не рекомендуется, так как это повлияет на все сеансы.

`lock_timeout` (`integer`)

Задаёт максимальную длительность ожидания (в миллисекундах) любым оператором получения блокировки таблицы, индекса, строки или другого объекта базы данных. Если ожидание не закончилось за указанное время, оператор прерывается. Это ограничение действует на каждую попытку получения блокировки по отдельности и применяется как к явным запросам блокировки (например, `LOCK TABLE` или `SELECT FOR UPDATE` без `NOWAIT`), так и к неявным. При значении, равном нулю (по умолчанию), этот контроль длительности отключается.

В отличие от `statement_timeout`, этот тайм-аут может произойти только при ожидании блокировки. Заметьте, что при ненулевом `statement_timeout` бессмысленно задавать в `lock_timeout` такое же или большее значение, так как тайм-аут оператора всегда будет происходить раньше. Если `log_min_error_statement` имеет значение `ERROR` или ниже, оператор, прерванный по тайм-ауту, будет записан в журнал.

Устанавливать значение `lock_timeout` в `postgresql.conf` не рекомендуется, так как это повлияет на все сеансы.

`idle_in_transaction_session_timeout` (`integer`)

Завершать любые сеансы, в которых открытая транзакция простаивает дольше заданного (в миллисекундах) времени. Это позволяет освободить все блокировки сеанса и вновь

задействовать слот подключения; также это позволяет очистить кортежи, видимые только для этой транзакции. Подробнее это описано в Разделе 23.1.

Значение по умолчанию (0) отключает это поведение.

`vacuum_freeze_table_age` (`integer`)

Задаёт максимальный возраст для поля `pg_class.relfrozenxid` таблицы, при достижении которого `VACUUM` будет производить агрессивное сканирование. Агрессивное сканирование отличается от обычного сканирования `VACUUM` тем, что затрагивает все страницы, которые могут содержать незамороженные XID или MXID, а не только те, что могут содержать мёртвые кортежи. Значение по умолчанию — 150 миллионов транзакций. Хотя пользователи могут задать любое значение от нуля до двух миллиардов, в `VACUUM` введён внутренний предел для действующего значения, равный 95% от autovacuum_freeze_max_age, чтобы периодически запускаемая вручную команда `VACUUM` имела шансы выполниться, прежде чем для таблицы будет запущена автоочистка для предотвращения зацикливания. За дополнительными сведениями обратитесь к Подразделу 23.1.5.

`vacuum_freeze_min_age` (`integer`)

Задаёт возраст для отсечки (в транзакциях), при достижении которого команда `VACUUM` должна замораживать версии строк при сканировании таблицы. Значение по умолчанию — 50 миллионов транзакций. Хотя пользователи могут задать любое значение от нуля до одного миллиарда, в `VACUUM` введён внутренний предел для действующего значения, равный половине autovacuum_freeze_max_age, чтобы принудительная автоочистка выполнялась не слишком часто. За дополнительными сведениями обратитесь к Подразделу 23.1.5.

`vacuum_multixact_freeze_table_age` (`integer`)

Задаёт максимальный возраст для поля `pg_class.relminmxid` таблицы, при достижении которого команда `VACUUM` будет выполнять агрессивное сканирование. Агрессивное сканирование отличается от обычного сканирования `VACUUM` тем, что затрагивает все страницы, которые могут содержать незамороженные XID или MXID, а не только те, что могут содержать мёртвые кортежи. Значение по умолчанию — 150 миллионов мультитранзакций. Хотя пользователи могут задать любое значение от нуля до двух миллиардов, в `VACUUM` введён внутренний предел для действующего значения, равный 95% от autovacuum_multixact_freeze_max_age, чтобы периодически запускаемая вручную команда `VACUUM` имела шансы выполниться, прежде чем для таблицы будет запущена автоочистка для предотвращения зацикливания. За дополнительными сведениями обратитесь к Подразделу 23.1.5.1.

`vacuum_multixact_freeze_min_age` (`integer`)

Задаёт возраст для отсечки (в мультитранзакциях), при достижении которого команда `VACUUM` должна заменять идентификаторы мультитранзакций новыми идентификаторами транзакций или мультитранзакций при сканировании таблицы. Значение по умолчанию — 5 миллионов мультитранзакций. Хотя пользователи могут задать любое значение от нуля до одного миллиарда, в `VACUUM` введён внутренний предел для действующего значения, равный половине autovacuum_multixact_freeze_max_age, чтобы принудительная автоочистка не выполнялась слишком часто. За дополнительными сведениями обратитесь к Подразделу 23.1.5.1.

`bytea_output` (`enum`)

Задаёт выходной формат для значения типа `bytea`. Это может быть формат `hex` (по умолчанию) или `escape` (традиционный формат Postgres Pro). За дополнительными сведениями обратитесь к Разделу 8.4. Входные значения `bytea` воспринимаются в обоих форматах, независимо от данного параметра.

`xmlbinary` (`enum`)

Задаёт способ кодирования двоичных данных в XML. Это кодирование применяется, например, когда значения `bytea` преобразуются в XML функциями `xmlelement` или `xmlforest`. Допустимые

варианты, определённые в стандарте XML-схем: base64 и hex. Значение по умолчанию — base64. Чтобы узнать больше о функциях для работы с XML, обратитесь к Разделу 9.14.

Конечный выбор в основном дело вкуса, ограничения могут накладываться только клиентскими приложениями. Оба метода поддерживают все возможные значения, хотя результат кодирования в base64 немного компактнее шестнадцатеричного вида (hex).

xmloption (enum)

Задаёт подразумеваемый по умолчанию тип преобразования между XML и символьными строками (DOCUMENT или CONTENT). За описанием этого преобразования обратитесь к Разделу 8.13. Значение по умолчанию — CONTENT (кроме него допускается значение DOCUMENT).

Согласно стандарту SQL этот параметр должен задаваться командой

```
SET XML OPTION { DOCUMENT | CONTENT };
```

Этот синтаксис тоже поддерживается в Postgres Pro.

gin_pending_list_limit (integer)

Задаёт максимальный размер очереди записей GIN, которая используется, когда включён режим fastupdate. Если размер очереди превышает заданный предел, записи из неё массово переносятся в основную структуру данных GIN, и очередь очищается. Размер по умолчанию — четыре мегабайта (4MB). Этот предел можно переопределить для отдельных индексов GIN, изменив их параметры хранения. За дополнительными сведениями обратитесь к Подразделу 60.4.1 и Разделу 60.5.

18.11.2. Языковая среда и форматы

DateStyle (string)

Задаёт формат вывода значений даты и времени, а также правила интерпретации неоднозначных значений даты. По историческим причинам эта переменная содержит два независимых компонента: указание выходного формата (ISO, Postgres, SQL и German) и указание порядка год(Y)/месяц(M)/день(D) для вводимых и выводимых значений (DMY, MDY или YMD). Эти два компонента могут задаваться по отдельности или вместе. Ключевые слова Euro и European являются синонимами DMY, а ключевые слова US, NonEuro и NonEuropean — синонимы MDY. За дополнительными сведениями обратитесь к Разделу 8.5. Встроенное значение по умолчанию — ISO, MDY, но initdb при инициализации записывает в файл конфигурации значение, соответствующее выбранной локали lc_time.

IntervalStyle (enum)

Задаёт формат вывода для значений-интервалов. В формате sql_standard интервал выводится в виде, установленном стандартом SQL. В формате postgres (выбранном по умолчанию) интервал выводится в виде, применявшемся в PostgreSQL до версии 8.4, когда параметр DateStyle имел значение ISO. В формате postgres_verbose интервал выводится в виде, применявшемся в PostgreSQL до версии 8.4, когда параметр DateStyle имел значение не ISO. В формате iso_8601 выводимая строка будет соответствовать «формату с кодами», определённому в разделе 4.4.3.2 стандарта ISO 8601.

На интерпретацию неоднозначных вводимых данных также влияет параметр IntervalStyle. За дополнительными сведениями обратитесь к Подразделу 8.5.4.

TimeZone (string)

Задаёт часовой пояс для вывода и ввода значений времени. Встроенное значение по умолчанию — GMT, но обычно оно переопределяется в postgresql.conf; initdb устанавливает в нём значение, соответствующее системному окружению. За дополнительными сведениями обратитесь к Подразделу 8.5.3.

`timezone_abbreviations` (`string`)

Задаёт набор сокращений часовых поясов, которые будут приниматься сервером во вводимых значениях даты и времени. Значение по умолчанию — `'Default'`, которое представляет набор основных сокращений, принятых в мире; допускаются также значения `'Australia'` и `'India'`, кроме них для конкретной инсталляции можно определить и другие наборы. За дополнительными сведениями обратитесь к Разделу B.4.

`extra_float_digits` (`integer`)

Этот параметр корректирует число цифр, выводимых при отображении чисел с плавающей точкой, включая типы `float4`, `float8` и геометрические типы. Значение параметра добавляется к стандартному числу цифр (`FLT_DIG` или `DBL_DIG`, в зависимости от типа). Значение может быть положительным, до 3, и тогда в выводе добавляются частично значимые цифры (это особенно полезно для выгрузки чисел с плавающей точкой, которые должны быть восстановлены точно), или отрицательным, тогда в выводе будут подавляться нежелательные цифры. См. также Подраздел 8.1.3.

`client_encoding` (`string`)

Задаёт кодировку (набор символов) на стороне клиента. По умолчанию выбирается кодировка базы данных. Наборы символов, которые поддерживает сервер Postgres Pro, перечислены в Подразделе 22.3.1.

`lc_messages` (`string`)

Устанавливает язык выводимых сообщений. Набор допустимых значений зависит от системы; за дополнительными сведениями обратитесь к Разделу 22.1. Если эта переменная определена как пустая строка (по умолчанию), то действующее значение получается из среды выполнения сервера, в зависимости от системы.

В некоторых системах такая категория локали отсутствует, так что даже если задать значение этой переменной, действовать оно не будет. Также может оказаться, что переведённые сообщения для запрошенного языка отсутствуют. В этих случаях вы по-прежнему будете получать сообщения на английском языке.

Изменить этот параметр могут только суперпользователи. Он влияет и на сообщения, которые сервер передаёт клиентам, и на те, что записываются в журнал, поэтому неподходящее значение может сделать серверные журналы нечитаемыми.

`lc_monetary` (`string`)

Устанавливает локаль для форматирования денежных сумм, например с использованием функций семейства `to_char`. Набор допустимых значений зависит от системы; за дополнительными сведениями обратитесь к Разделу 22.1. Если эта переменная определена как пустая строка (по умолчанию), то действующее значение получается из среды выполнения сервера, в зависимости от системы.

`lc_numeric` (`string`)

Устанавливает локаль для форматирования чисел, например с использованием функций семейства `to_char`. Набор допустимых значений зависит от системы; за дополнительными сведениями обратитесь к Разделу 22.1. Если эта переменная определена как пустая строка (по умолчанию), то действующее значение получается из среды выполнения сервера, в зависимости от системы.

`lc_time` (`string`)

Устанавливает локаль для форматирования даты и времени, например с использованием функций семейства `to_char`. Набор допустимых значений зависит от системы; за дополнительными сведениями обратитесь к Разделу 22.1. Если эта переменная определена как пустая строка (по умолчанию), то действующее значение получается из среды выполнения сервера, в зависимости от системы.

`default_text_search_config` (`string`)

Выбирает конфигурацию текстового поиска для тех функций текстового поиска, которым не передаётся аргумент, явно указывающий конфигурацию. За дополнительной информацией обратитесь к Главе 12. Встроенное значение по умолчанию — `pg_catalog.simple`, но initdb при инициализации записывает в файл конфигурации сервера значение, соответствующее выбранной локали `lc_ctype`, если удастся найти такую конфигурацию текстового поиска.

18.11.3. Предзагрузка разделяемых библиотек

Для настройки предварительной загрузки разделяемых библиотек в память сервера, в целях подключения дополнительной функциональности или увеличения быстродействия, предназначены несколько параметров. Значения этих параметров задаются однотипно, например, со значением `'$libdir/mylib'` в память будет загружена `mylib.so` (или в некоторых ОС, `mylib.sl`) из стандартного каталога библиотек данной инсталляции сервера. Различаются эти параметры тем, когда они вступают в силу и какие права требуются для их изменения.

Таким же образом можно загрузить библиотеки на процедурных языках Postgres Pro, обычно в виде `'$libdir/plXXX'`, где `XXX` — имя языка: `pgsql`, `perl`, `tcl` или `python`.

Этим способом можно загрузить только разделяемые библиотеки, предназначенные специально для использования с Postgres Pro. Postgres Pro при загрузке библиотеки проверяет наличие «отличительного блока» для гарантии совместимости. Поэтому загрузить библиотеки не для Postgres Pro таким образом нельзя. Для этого вы можете воспользоваться средствами операционной системы, например, переменной окружения `LD_PRELOAD`.

В общем случае, чтобы узнать, какой способ рекомендуется для загрузки модуля, следует обратиться к документации этого модуля.

`local_preload_libraries` (`string`)

В этом параметре задаются одна или несколько разделяемых библиотек, которые будут загружаться при установлении соединения. Он содержит разделённый запятыми список имён библиотек, и каждое имя в нём должно восприниматься командой LOAD. Пробельные символы между именами игнорируются; если в имя нужно включить пробелы или запятые, заключите его в двойные кавычки. Заданное значение параметра действует только в начале соединения, так что последующие изменения ни на что не влияют. Если указанная в нём библиотека не найдена, установить подключение не удастся.

Этот параметр разрешено устанавливать всем пользователям. Поэтому библиотеки, которые так можно загрузить, ограничиваются теми, что находятся в подкаталоге `plugins` стандартного каталога библиотек установленного сервера. (Ответственность за то, чтобы в этом подкаталоге находились только «безопасные» библиотеки, лежит на администраторе.) В `local_preload_libraries` этот каталог можно задать явно (например, так: `$libdir/plugins/mylib`), либо просто указать имя библиотеки — `mylib` (оно будет воспринято как `$libdir/plugins/mylib`).

Данный механизм предназначен для того, чтобы непривилегированные пользователи могли загружать отладочные или профилирующие библиотеки в избранных сеансах, обходясь без явной команды `LOAD`. Для такого применения этот параметр обычно устанавливается в переменной окружения `PGOPTIONS` на клиенте или с помощью команды `ALTER ROLE SET`.

Обычно этот параметр не следует использовать, если только модуль не предназначен специально для такой загрузки обычными пользователями. Предпочтительная альтернатива ему — session_preload_libraries.

`session_preload_libraries` (`string`)

В этом параметре задаются одна или несколько разделяемых библиотек, которые будут загружаться при установлении соединения. Он содержит разделённый запятыми список имён библиотек, и каждое имя в нём должно восприниматься командой LOAD. Пробельные символы

между именами игнорируются; если в имя нужно включить пробелы или запятые, заключите его в двойные кавычки. Заданное значение параметра действует только в начале соединения, так что последующие изменения ни на что не влияют. Если указанная в нём библиотека не найдена, установить подключение не удастся. Изменить его могут только суперпользователи.

Данный параметр предназначен для загрузки отладочных или профилирующих библиотек в избранных сеансах, без явного выполнения команды `LOAD`. Например, можно загрузить модуль auto_explain во всех сеансах пользователя с заданным именем, установив этот параметр командой `ALTER ROLE SET`. Кроме того, этот параметр можно изменить без перезапуска сервера (хотя изменения вступают в силу только при запуске нового сеанса), так что таким образом проще подгружать новые модули, даже если это нужно сделать для всех сеансов.

В отличие от shared_preload_libraries, этот вариант загрузки библиотеки не даёт большого выигрыша в скорости по сравнению с вариантом загрузки при первом использовании. Однако он оказывается выигрышным, когда используется пул соединений.

`shared_preload_libraries` (`string`)

В этом параметре задаются одна или несколько разделяемых библиотек, которые будут загружаться при запуске сервера. Он содержит разделённый запятыми список имён библиотек, и каждое имя в нём должно восприниматься командой LOAD. Пробельные символы между именами игнорируются; если в имя нужно включить пробелы или запятые, заключите его в двойные кавычки. Этот параметр можно задать только при запуске сервера. Если указанная в нём библиотека не будет найдена, сервер не запустится.

Некоторые библиотеки при загрузке должны выполнять операции, которые могут иметь место только при запуске главного процесса, например, выделять разделяемую память, резервировать легковесные блокировки или запускать фоновые рабочие процессы. Такие библиотеки должны загружаться при запуске сервера посредством этого параметра. За подробностями обратитесь к документации библиотек.

Также можно предварительно загрузить и другие библиотеки. Предварительная загрузка позволяет избавиться от задержки, возникающей при первом использовании библиотеки. Однако при этом может несколько увеличиться время запуска каждого нового процесса, даже если он не будет использовать эту библиотеку. Поэтому применять этот параметр рекомендуется только для библиотек, которые будут использоваться большинством сеансов. Кроме того, при изменении этого параметра необходимо перезапускать сервер, так что этот вариант не подходит, например, для краткосрочных задач отладки. В таких случаях используйте вместо него session_preload_libraries.

> **Примечание**
>
> В системе Windows загрузка библиотек при запуске сервера не сокращает время запуска каждого нового серверного процесса; каждый процесс будет заново загружать все библиотеки. Однако параметр `shared_preload_libraries` всё же может быть полезен в Windows для загрузки библиотек, которые должны выполнять некоторые операции при запуске главного процесса.

18.11.4. Другие параметры по умолчанию

`dynamic_library_path` (`string`)

Когда требуется открыть динамически загружаемый модуль и его имя, заданное в команде `CREATE FUNCTION` или `LOAD` не содержит имён каталогов (т. е. в этом имени нет косой черты), система будет искать запрошенный файл в данном пути.

Значением параметра `dynamic_library_path` должен быть список абсолютных путей, разделённых двоеточием (или точкой с запятой в Windows). Если элемент в этом списке

начинается со специальной строки `$libdir`, вместо неё подставляется заданный при компиляции путь каталога библиотек Postgres Pro; в этот каталог устанавливаются модули, поставляемые в составе стандартного дистрибутива Postgres Pro. (Чтобы узнать имя этого каталога, можно выполнить `pg_config --pkglibdir`.) Например:

```
dynamic_library_path = '/usr/local/lib/postgresql:/home/my_project/lib:$libdir'
```

Или в среде Windows:

```
dynamic_library_path = 'C:\tools\postgresql;H:\my_project\lib;$libdir'
```

Значение по умолчанию этого параметра — `'$libdir'`. Если его значение — пустая строка, автоматический поиск по заданному пути отключается.

Суперпользователи могут изменить этот параметр в процессе работы сервера, но такое изменение будет действовать только до завершения клиентского соединения, так что этот вариант следует оставить для целей разработки. Для других целей этот параметр рекомендуется устанавливать в файле конфигурации `postgresql.conf`.

`gin_fuzzy_search_limit` (`integer`)

Задаёт мягкий верхний лимит для размера набора, возвращаемого при сканировании индексов GIN. За дополнительными сведениями обратитесь к Разделу 60.5.

18.12. Управление блокировками

`deadlock_timeout` (`integer`)

Время ожидания блокировки (в миллисекундах), по истечении которого будет выполняться проверка состояния взаимоблокировки. Эта проверка довольно дорогостоящая, поэтому сервер не выполняет её при всяком ожидании блокировки. Мы оптимистично полагаем, что взаимоблокировки редки в производственных приложениях, и поэтому просто ждём некоторое время, прежде чем пытаться выявить взаимоблокировку. При увеличении значения этого параметра сокращается время, уходящее на ненужные проверки взаимоблокировки, но замедляется реакция на реальные взаимоблокировки. Значение по умолчанию — одна секунда (`1s`), что близко к минимальному значению, которое стоит применять на практике. На сервере с большой нагрузкой имеет смысл увеличить его. В идеале это значение должно превышать типичное время транзакции, чтобы повысить шансы на то, что блокировка всё-таки будет освобождена, прежде чем ожидающая транзакция решит проверить состояние взаимоблокировки. Изменить этот параметр могут только суперпользователи.

Когда включён параметр log_lock_waits, данный параметр также определяет, спустя какое время в журнал сервера будут записываться сообщения об ожидании блокировки. Если вы пытаетесь исследовать задержки, вызванные блокировками, имеет смысл уменьшить его по сравнению с обычным значением `deadlock_timeout`.

`max_locks_per_transaction` (`integer`)

Общая таблица блокировок отслеживает блокировки для `max_locks_per_transaction` * (max_connections + max_prepared_transactions) объектов (например, таблиц); таким образом, в любой момент времени может быть заблокировано не больше этого числа различных объектов. Этот параметр управляет средним числом блокировок объектов, выделяемым для каждой транзакции; отдельные транзакции могут заблокировать и больше объектов, если все они умещаются в таблице блокировок. Заметьте, что это *не* число строк, которое может быть заблокировано; их количество не ограничено. Значение по умолчанию, 64, как показала практика, вполне приемлемо, но может возникнуть потребность его увеличить, если запросы обращаются ко множеству различных таблиц в одной транзакции, как например, запрос к родительской таблице со многими потомками. Этот параметр можно задать только при запуске сервера.

Для ведомого сервера значение этого параметра должно быть больше или равно значению на ведущем. В противном случае на ведомом сервере не будут разрешены запросы.

max_pred_locks_per_transaction (integer)

Общая таблица предикатных блокировок отслеживает блокировки для max_pred_locks_per_transaction * (max_connections + max_prepared_transactions) объектов (например, таблиц); таким образом, в один момент времени может быть заблокировано не больше этого числа различных объектов. Этот параметр управляет средним числом блокировок объектов, выделяемым для каждой транзакции; отдельные транзакции могут заблокировать и больше объектов, если все они умещаются в таблице блокировок. Заметьте, что это *не* число строк, которое может быть заблокировано; их количество не ограничено. Значение по умолчанию, 64, как показала практика, вполне приемлемо, но может возникнуть потребность его увеличить, если запросы обращаются ко множеству различных таблиц в одной сериализуемой транзакции, как например, запрос к родительской таблице со многими потомками. Этот параметр можно задать только при запуске сервера.

max_pred_locks_per_relation (integer)

Этот параметр определяет, для скольких страниц или кортежей одного отношения могут устанавливаться предикатные блокировки, прежде чем вместо них будет затребована одна блокировка для всего отношения. Значения, большие или равные нулю, задают абсолютный предел, а с отрицательным значением пределом будет значение max_pred_locks_per_transaction, делённое на модуль данного. По умолчанию действует значение -2, что даёт то же поведение, что наблюдалось в предыдущих версиях Postgres Pro. Этот параметр можно задать только в файле postgresql.conf или в командной строке при запуске сервера.

max_pred_locks_per_page (integer)

Этот параметр определяет, для скольких строк на одной странице могут устанавливаться предикатные блокировки, прежде чем вместо них будет затребована одна блокировка для всей страницы. Этот параметр можно задать только в файле postgresql.conf или в командной строке при запуске сервера.

18.13. Совместимость с разными версиями и платформами

18.13.1. Предыдущие версии Postgres Pro

array_nulls (boolean)

Этот параметр определяет, будет ли при разборе вводимого массива распознаваться строка NULL без кавычек как элемент массива, равный NULL. Значение по умолчанию, on, позволяет задавать NULL в качестве элементов вводимого массива. Однако до версии 8.2 PostgreSQL не поддерживал ввод элементов NULL в массивах, а воспринимал NULL как обычный элемент массива со строковым значением «NULL». Для обратной совместимости с приложениями, зависящими от старого поведения, эту переменную можно отключить (присвоив ей off).

Заметьте, что массивы, содержащие NULL, можно создать, даже когда эта переменная имеет значение off.

backslash_quote (enum)

Этот параметр определяет, можно ли будет представить знак апострофа в строковой константе в виде \'. В стандарте SQL определён другой, предпочитаемый вариант передачи апострофа, дублированием (''), но Postgres Pro исторически также принимал вариант \'. Однако применение варианта \' сопряжено с угрозами безопасности, так как в некоторых клиентских кодировках существуют многобайтные символы, последний байт которых численно равен ASCII-коду \. Если код на стороне клиента выполнит экранирование некорректно, это может открыть возможности для SQL-инъекции. Предотвратить этот риск можно, запретив серверу принимать запросы, в которых апостроф экранируется обратной косой. Допустимые значения параметра backslash_quote: on (принимать \' всегда), off (не принимать никогда) и

safe_encoding (принимать, только если клиентская кодировка не допускает присутствия ASCII-кода \ в многобайтных символах). Значение по умолчанию — safe_encoding.

Заметьте, что в строковой константе, записанной согласно стандарту, знаки \ обозначают просто \. Этот параметр влияет только на восприятие строк, не соответствующих стандарту, в том числе с синтаксисом спецпоследовательностей (E'...').

default_with_oids (boolean)

Этот параметр определяет, будут ли команды CREATE TABLE и CREATE TABLE AS без явных указаний WITH OIDS и WITHOUT OIDS добавлять столбец OID в создаваемые таблицы. Он также устанавливает, будут ли столбцы OID добавляться в таблицы, создаваемые командой SELECT INTO. По умолчанию значение этого параметра — off (столбцы OID не добавляются); в PostgreSQL версии 8.0 и ранее он был включён (on).

Практика использования OID в пользовательских таблицах считается устаревшей, так что в большинстве инсталляций не следует включать этот параметр. Приложения, которым требуется столбец OID в определённой таблице, могут явно указать WITH OIDS при создании таблицы. Этот параметр следует включать только для совместимости со старыми приложениями, которые не делают этого.

escape_string_warning (boolean)

Когда этот параметр включён, сервер выдаёт предупреждение, если обратная косая черта (\) встречается в обычной строковой константе (с синтаксисом '...') и параметр standard_conforming_strings отключён. Значение по умолчанию — on (вкл.).

Приложения, которые предпочитают использовать обратную косую в виде спецсимвола, должны перейти к применению синтаксиса спецстрок (E'...'), так как по умолчанию теперь в обычных строках обратная косая воспринимается как обычный символ, в соответствии со стандартом SQL. Включение данного параметра помогает найти код, нуждающийся в модификации.

lo_compat_privileges (boolean)

В PostgreSQL до версии 9.0 для больших объектов не назначались права доступа, и поэтому они были всегда доступны на чтение и запись для всех пользователей. Если установить для этого параметра значение on, существующие теперь проверки прав отключаются для совместимости с предыдущими версиями. Значение по умолчанию — off. Изменить этот параметр могут только суперпользователи.

Установка данного параметра не приводит к отключению всех проверок безопасности, связанных с большими объектами — затрагиваются только те проверки, которые изменились в PostgreSQL 9.0. Например, функции lo_import() и lo_export() будут требовать прав суперпользователя вне зависимости от данного значения.

operator_precedence_warning (boolean)

Когда этот параметр включён, анализатор запроса будет выдавать предупреждение для всех конструкций, которые поменяли поведение после PostgreSQL 9.4 в результате изменения приоритетов операторов. Это полезно для аудита, так как позволяет понять, не сломалось ли что-то вследствие этого изменения. Но в производственной среде включать его не следует, так как предупреждения могут выдаваться и тогда, когда код абсолютно правильный и соответствует стандарту SQL. Значение по умолчанию — off.

За подробностями обратитесь к Подразделу 4.1.6.

quote_all_identifiers (boolean)

Принудительно заключать в кавычки все идентификаторы, даже если это не ключевые слова (сегодня), при получении SQL из базы данных. Это касается вывода EXPLAIN, а также

результатов функций типа pg_get_viewdef. См. также описание аргумента --quote-all-identifiers команд pg_dump и pg_dumpall.

nul_byte_replacement_on_import (string)

Заменять нулевые байты ('\0') ASCII-символом с указанным десятичным кодом при загрузке данных с использованием команды COPY FROM. Такая замена может потребоваться при загрузке данных из другой СУБД, так как Postgres Pro не принимает нулевые байты в данных. ASCII-символ замены должен отличаться от символов QUOTE и DELIMITER, применяемых при выполнении COPY FROM; в противном случае результат может быть неожиданным. Значение по умолчанию — '\0', то есть замена не производится.

standard_conforming_strings (boolean)

Этот параметр определяет, будет ли обратная косая черта в обычных строковых константах ('...') восприниматься буквально, как того требует стандарт SQL. Начиная с версии PostgreSQL 9.1, он имеет значение on (в предыдущих версиях значение по умолчанию было off). Приложения могут выяснить, как обрабатываются строковые константы, проверив этот параметр. Наличие этого параметра может также быть признаком того, что поддерживается синтаксис спецпоследовательностей (E'...'). Этот синтаксис (Подраздел 4.1.2.2) следует использовать, если приложению нужно, чтобы обратная косая воспринималась как спецсимвол.

synchronize_seqscans (boolean)

Этот параметр включает синхронизацию обращений при последовательном сканировании больших таблиц, чтобы эти операции читали один блок примерно в одно и то же время, и, таким образом, нагрузка разделялась между ними. Когда он включён, сканирование может начаться в середине таблицы, чтобы синхронизироваться со сканированием, которое уже выполняется. По достижении конца таблицы сканирование «заворачивается» к началу и завершает обработку пропущенных строк. Это может привести к непредсказуемому изменению порядка строк, возвращаемых запросами, в которых отсутствует предложение ORDER BY. Когда этот параметр выключен (имеет значение off), реализуется поведение, принятое до версии 8.3, когда последовательное сканирование всегда начиналось с начала таблицы. Значение по умолчанию — on.

18.13.2. Совместимость с разными платформами и клиентами

transform_null_equals (boolean)

Когда этот параметр включён, проверки вида *выражение* = NULL (или NULL = *выражение*) воспринимаются как *выражение* IS NULL, то есть они истинны, если *выражение* даёт значение NULL, и ложны в противном случае. Согласно спецификации SQL, сравнение *выражение* = NULL должно всегда возвращать NULL (неизвестное значение). Поэтому по умолчанию этот параметр выключен (равен off).

Однако формы фильтров в Microsoft Access генерируют запросы, в которых проверка на значение NULL записывается как *выражение* = NULL, так что если вы используете этот интерфейс для обращения к базе данных, имеет смысл включить данный параметр. Так как проверки вида *выражение* = NULL всегда возвращают значение NULL (следуя правилам стандарта SQL), они не очень полезны и не должны встречаться в обычных приложениях, так что на практике от включения этого параметра не будет большого вреда. Однако начинающие пользователи часто путаются в семантике выражений со значениями NULL, поэтому по умолчанию этот параметр выключен.

Заметьте, что этот параметр влияет только на точную форму сравнения = NULL, но не на другие операторы сравнения или выражения, результат которых может быть равнозначен сравнению с применением оператора равенства (например, конструкцию IN). Поэтому данный параметр не может быть универсальной защитой от плохих приёмов программирования.

За сопутствующей информацией обратитесь к Разделу 9.2.

18.14. Обработка ошибок

`exit_on_error` (`boolean`)

Если этот параметр включён, любая ошибка приведёт к прерыванию текущего сеанса. По умолчанию он отключён, так что сеанс будет прерываться только при критических ошибках.

`restart_after_crash` (`boolean`)

Когда этот параметр включён (это состояние по умолчанию), Postgres Pro будет автоматически перезагружаться после сбоя серверного процесса. Такой вариант позволяет обеспечить максимальную степень доступности базы данных. Однако в некоторых обстоятельствах, например, когда Postgres Pro управляется кластерным ПО, такую перезагрузку лучше отключить, чтобы кластерное ПО могло вмешаться и выполнить, возможно, более подходящие действия.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`data_sync_retry` (`boolean`)

При выключенном значении этого параметра (по умолчанию) Postgres Pro будет выдавать ошибку уровня PANIC в случае неудачи при попытке сохранить изменённые данные в файловой системе. В результате сервер баз данных остановится аварийно. Задать этот параметр можно только при запуске сервера.

В некоторых операционных системах состояние данных в кеше внутри ядра оказывается неопределённым при ошибке записи. В каких-то случаях эти данные могут быть просто утеряны, и повторять попытку записи небезопасно: вторая попытка может оказаться успешной, тогда как на деле данные не сохранены. В этих обстоятельствах единственный способ избежать потери данных — восстановить их из WAL после такого сбоя, но перед этим желательно выяснить причину проблемы и, возможно, заменить нерабочее оборудование.

Если включить этот параметр, Postgres Pro в случае сбоя при записи выдаст ошибку, но продолжит работу в расчёте повторить операцию сохранения данных при последующей контрольной точке. Включать его следует, только если достоверно известно, как поступает система с данными в буфере при ошибке записи.

18.15. Предопределённые параметры

Следующие «параметры» доступны только для чтения, их значения задаются при компиляции или при установке Postgres Pro. По этой причине они исключены из примера файла `postgresql.conf`. Эти параметры сообщают различные аспекты поведения Postgres Pro, которые могут быть интересны для определённых приложений, например, средств администрирования.

`block_size` (`integer`)

Сообщает размер блока на диске. Он определяется значением `BLCKSZ` при сборке сервера. Значение по умолчанию — 8192 байта. Значение `block_size` влияет на некоторые другие переменные конфигурации (например, shared_buffers). Об этом говорится в Разделе 18.4.

`data_checksums` (`boolean`)

Сообщает, включён ли в этом кластере контроль целостности данных. За дополнительными сведениями обратитесь к Контрольные суммы данных.

`debug_assertions` (`boolean`)

Сообщает, был ли Postgres Pro собран с проверочными утверждениями. Это имеет место, когда при сборке Postgres Pro определяется макрос `USE_ASSERT_CHECKING` (например, при выполнении `configure` с флагом `--enable-cassert`). По умолчанию Postgres Pro собирается без проверочных утверждений.

integer_datetimes (boolean)

Сообщает, был ли Postgres Pro собран с поддержкой даты и времени в 64-битных целых. Начиная с Postgres Pro версии 10, он всегда равен on.

lc_collate (string)

Сообщает локаль, по правилам которой выполняется сортировка текстовых данных. За дополнительными сведениями обратитесь к Разделу 22.1. Это значение определяется при создании базы данных.

lc_ctype (string)

Сообщает локаль, определяющую классификацию символов. За дополнительными сведениями обратитесь к Разделу 22.1. Это значение определяется при создании базы данных. Обычно оно не отличается от lc_collate, но для некоторых приложений оно может быть определено по-другому.

max_function_args (integer)

Сообщает верхний предел для числа аргументов функции. Он определяется константой FUNC_MAX_ARGS при сборке сервера. По умолчанию установлен предел в 100 аргументов.

max_identifier_length (integer)

Сообщает максимальную длину идентификатора. Она определяется числом на 1 меньше, чем NAMEDATALEN, при сборке сервера. По умолчанию константа NAMEDATALEN равна 64; следовательно max_identifier_length по умолчанию равна 63 байтам, но число символов в многобайтной кодировке будет меньше.

max_index_keys (integer)

Сообщает верхний предел для числа ключей индекса. Он определяется константой INDEX_MAX_KEYS при сборке сервера. По умолчанию установлен предел в 32 ключа.

segment_size (integer)

Сообщает, сколько блоков (страниц) можно сохранить в одном файловом сегменте. Это число определяется константой RELSEG_SIZE при сборке сервера. Максимальный размер сегмента в файлах равен произведению segment_size и block_size; по умолчанию это 1 гигабайт.

server_encoding (string)

Сообщает кодировку базы данных (набор символов). Она определяется при создании базы данных. Обычно клиентов должно интересовать только значение client_encoding.

server_version (string)

Сообщает номер версии сервера. Она определяется константой PG_VERSION при сборке сервера.

server_version_num (integer)

Сообщает номер версии сервера в виде целого числа. Она определяется константой PG_VERSION_NUM при сборке сервера.

wal_block_size (integer)

Сообщает размер блока WAL на диске. Он определяется константой XLOG_BLCKSZ при сборке сервера. Значение по умолчанию — 8192 байта.

wal_segment_size (integer)

Сообщает число блоков (страниц) в файле сегмента WAL. Общий размер файла сегмента WAL равняется произведению wal_segment_size и wal_block_size; по умолчанию это 16 мегабайт. За дополнительными сведениями обратитесь к Разделу 29.4.

18.16. Внесистемные параметры

Поддержка внесистемных параметров была реализована, чтобы дополнительные модули (например, процедурные языки) могли добавлять собственные параметры, неизвестные серверу Postgres Pro. Это позволяет единообразно настраивать модули расширения.

Имена параметров расширений записываются следующим образом: имя расширения, точка и затем собственно имя параметра, подобно полным именам объектов в SQL. Например: `plpgsql.variable_conflict`.

Так как внесистемные параметры могут быть установлены в процессах, не загружающих соответствующий модуль расширения, Postgres Pro принимает значения для любых имён с двумя компонентами. Такие параметры воспринимаются как заготовки и не действуют до тех пор, пока не будет загружен определяющий их модуль. Когда модуль расширения загружается, он добавляет свои определения параметров и присваивает все заготовленные значения этим параметрам, либо выдаёт предупреждение, если начинающееся с имени данного расширения имя заготовленного параметра оказывается нераспознанным.

18.17. Параметры для разработчиков

Следующие параметры предназначены для работы над исходным кодом Postgres Pro, а в некоторых случаях они могут помочь восстановить сильно повреждённые базы данных. Для использования их в производственной среде не должно быть причин, поэтому они исключены из примера файла `postgresql.conf`. Заметьте, что для работы со многими из этих параметров требуются специальные флаги компиляции.

`allow_in_place_tablespaces` (`boolean`)

Позволяет создавать табличные пространства как каталоги в `pg_tblspc`, когда в команде `CREATE TABLESPACE` не указано расположение каталога. Это предназначено для тестирования сценариев репликации, когда ведущий и ведомый серверы работают на одном компьютере. Средства резервного копирования ожидают получить символические ссылки в этом месте, поэтому их работа может нарушиться. Изменять этот параметр могут только суперпользователи.

`allow_system_table_mods` (`boolean`)

Разрешает модификацию структуры системных таблиц. Этот параметр используется командой `initdb`. Задать этот параметр можно только при запуске сервера.

`ignore_system_indexes` (`boolean`)

Отключает использование индексов при чтении системных таблиц (при этом индексы всё же будут изменяться при записи в эти таблицы). Это полезно для восстановления работоспособности при повреждённых системных индексах. Этот параметр нельзя изменить после запуска сеанса.

`post_auth_delay` (`integer`)

При ненулевом значении этот параметр задаёт задержку (в секундах) при запуске нового серверного процесса после выполнения процедуры аутентификации. Он предназначен для того, чтобы разработчики имели возможность подключить отладчик к серверному процессу. Этот параметр нельзя изменить после начала сеанса.

`pre_auth_delay` (`integer`)

При ненулевом значении этот параметр задаёт задержку (в секундах), добавляемую сразу после порождения нового серверного процесса, до выполнения процедуры аутентификации. Он предназначен для того, чтобы разработчики имели возможность подключить отладчик к серверному процессу при решении проблем с аутентификацией. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`trace_notify` (`boolean`)

Включает вывод очень подробной отладочной информации при выполнении команд `LISTEN` и `NOTIFY`. Чтобы эти сообщения передавались клиенту или в журнал сервера, параметр client_min_messages или log_min_messages, соответственно, должен иметь значение `DEBUG1` или ниже.

`trace_recovery_messages` (`enum`)

Включает вывод в журнал отладочных сообщений, связанных с восстановлением, которые иначе не выводятся. Этот параметр позволяет пользователю переопределить обычное значение log_min_messages, но только для специфических сообщений. Он предназначен для отладки режима горячего резерва. Допустимые значения: `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1` и `LOG`. Значение по умолчанию, `LOG`, никак не влияет на запись этих сообщений в журнал. С другими значениями отладочные сообщения, связанные с восстановлением, имеющие заданный приоритет или выше, выводятся, как если бы они имели приоритет `LOG`; при стандартных значениях `log_min_messages` это означает, что они будут фиксироваться в журнале сервера. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`trace_sort` (`boolean`)

Включает вывод информации об использовании ресурсов во время операций сортировки. Этот параметр доступен, только если при сборке Postgres Pro был определён макрос `TRACE_SORT`. (По умолчанию макрос `TRACE_SORT` определён.)

`trace_locks` (`boolean`)

Включает вывод подробных сведений о блокировках. В эти сведения входит вид операции блокировки, тип блокировки и уникальный идентификатор объекта, который блокируется или разблокируется. Кроме того, в их составе выводятся битовые маски для типов блокировок, уже полученных для данного объекта, и для типов блокировок, ожидающих его освобождения. В дополнение к этому выводится количество полученных и ожидающих блокировок для каждого типа блокировок, а также их общее количество. Ниже показан пример вывода в журнал:

```
LOG:  LockAcquire: new: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  GrantLock: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(2) req(1,0,0,0,0,0,0)=1 grant(1,0,0,0,0,0,0)=1
      wait(0) type(AccessShareLock)
LOG:  UnGrantLock: updated: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  CleanUpLock: deleting: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(INVALID)
```

Подробнее о структуре выводимой информации можно узнать в `src/include/storage/lock.h`.

Этот параметр доступен, только если при компиляции Postgres Pro был определён макрос `LOCK_DEBUG`.

`trace_lwlocks` (`boolean`)

Включает вывод информации об использовании легковесных блокировок. Такие блокировки предназначены в основном для взаимоисключающего доступа к общим структурам данных в памяти.

Этот параметр доступен, только если при компиляции Postgres Pro был определён макрос `LOCK_DEBUG`.

`trace_userlocks` (`boolean`)

Включает вывод информации об использовании пользовательских блокировок. Она выводится в том же формате, что и с `trace_locks`, но по рекомендательным блокировкам.

Этот параметр доступен, только если при компиляции Postgres Pro был определён макрос `LOCK_DEBUG`.

`trace_lock_oidmin` (`integer`)

Если этот параметр установлен, при трассировке блокировок не будут отслеживаться таблицы с OID меньше заданного (это используется для исключения из трассировки системных таблиц).

Этот параметр доступен, только если при компиляции Postgres Pro был определён макрос `LOCK_DEBUG`.

`trace_lock_table` (`integer`)

Безусловно трассировать блокировки для таблицы с заданным OID.

Этот параметр доступен, только если при компиляции Postgres Pro был определён макрос `LOCK_DEBUG`.

`debug_deadlocks` (`boolean`)

Включает вывод информации обо всех текущих блокировках при тайм-ауте взаимоблокировки.

Этот параметр доступен, только если при компиляции Postgres Pro был определён макрос `LOCK_DEBUG`.

`log_btree_build_stats` (`boolean`)

Включает вывод статистики использования системных ресурсов (памяти и процессора) при различных операциях с B-деревом.

Этот параметр доступен, только если при компиляции Postgres Pro был определён макрос `BTREE_BUILD_STATS`.

`wal_consistency_checking` (`string`)

Этот параметр предназначен для проверки ошибок в процедурах воспроизведения WAL. Когда он включён, в записи WAL добавляются полные образы страниц всех изменяемых буферов, Когда запись впоследствии воспроизводится, система сначала применяет эту запись, а затем проверяет, соответствуют ли буферы, изменённые записью, сохранённым образам. В определённых случаях (например, во вспомогательных битах) небольшие изменения допускаются и будут игнорироваться. Если выявляются неожиданные различия, это считается критической ошибкой и восстановление прерывается.

По умолчанию значение этого параметра — пустая строка, так что эта функциональность отключена. Его значением может быть `all` (будут проверяться все записи) или список имён менеджеров ресурсов через запятую (будут проверяться записи, выдаваемые этими менеджерами). В настоящее время поддерживаются менеджеры `heap`, `heap2`, `btree`, `hash`, `gin`, `gist`, `sequence`, `spgist`, `brin` и `generic`. Изменять этот параметр могут только суперпользователи.

`wal_debug` (`boolean`)

Включает вывод отладочной информации, связанной с WAL. Этот параметр доступен, только если при компиляции Postgres Pro был определён макрос `WAL_DEBUG`.

`ignore_checksum_failure` (`boolean`)

Этот параметр действует, только если включён Контрольные суммы данных.

При обнаружении ошибок контрольных сумм при чтении Postgres Pro обычно сообщает об ошибке и прерывает текущую транзакцию. Если параметр `ignore_checksum_failure` включён, система игнорирует проблему (но всё же предупреждает о ней) и продолжает обработку. Это поведение может *привести к краху, распространению или сокрытию повреждения данных и другим серьёзными проблемам*. Однако включив его, вы можете обойти ошибку и получить неповреждённые данные, которые могут находиться в таблице, если цел заголовок блока. Если же повреждён заголовок, будет выдана ошибка, даже когда этот параметр включён. По умолчанию этот параметр отключён (имеет значение `off`) и изменить его состояние может только суперпользователь.

`zero_damaged_pages` (`boolean`)

При выявлении повреждённого заголовка страницы Postgres Pro обычно сообщает об ошибке и прерывает текущую транзакцию. Если параметр `zero_damaged_pages` включён, вместо этого система выдаёт предупреждение, обнуляет повреждённую страницу в памяти и продолжает обработку. Это поведение *разрушает данные*, а именно все строки в повреждённой странице. Однако включив его, вы можете обойти ошибку и получить строки из неповреждённых страниц, которые могут находиться в таблице. Это бывает полезно для восстановления данных, испорченных в результате аппаратной или программной ошибки. Обычно включать его следует только тогда, когда не осталось никакой другой надежды на восстановление данных в повреждённых страницах таблицы. Обнулённые страницы не сохраняются на диск, поэтому прежде чем выключать этот параметр, рекомендуется пересоздать проблемные таблицы или индексы. По умолчанию этот параметр отключён (имеет значение `off`) и изменить его состояние может только суперпользователь.

18.18. Краткие аргументы

Для удобства с некоторыми параметрами сопоставлены однобуквенные аргументы командной строки. Все они описаны в Таблице 18.3. Некоторые из этих сопоставлений существуют по историческим причинам, так что наличие однобуквенного синонима не обязательно является признаком того, что этот аргумент часто используется.

Таблица 18.3. Ключ краткого аргумента

Краткий аргумент	Эквивалент
`-B` *x*	`shared_buffers =` *x*
`-d` *x*	`log_min_messages = DEBUG` *x*
`-e`	`datestyle = euro`
`-fb`, `-fh`, `-fi`, `-fm`, `-fn`, `-fo`, `-fs`, `-ft`	`enable_bitmapscan = off`, `enable_hashjoin = off`, `enable_indexscan = off`, `enable_mergejoin = off`, `enable_nestloop = off`, `enable_indexonlyscan = off`, `enable_seqscan = off`, `enable_tidscan = off`
`-F`	`fsync = off`
`-h` *x*	`listen_addresses =` *x*
`-i`	`listen_addresses = '*'`
`-k` *x*	`unix_socket_directories =` *x*
`-l`	`ssl = on`
`-N` *x*	`max_connections =` *x*
`-O`	`allow_system_table_mods = on`
`-p` *x*	`port =` *x*
`-P`	`ignore_system_indexes = on`
`-s`	`log_statement_stats = on`

Краткий аргумент	**Эквивалент**
`-S` *x*	`work_mem =` *x*
`-tpa`, `-tpl`, `-te`	`log_parser_stats = on` , `log_planner_stats = on`, `log_executor_stats = on`
`-W` *x*	`post_auth_delay =` *x*

Глава 19. Аутентификация клиентского приложения

При подключении к серверу базы данных, клиентское приложение указывает имя пользователя Postgres Pro, так же как и при обычном входе пользователя на компьютер с ОС Unix. При работе в среде SQL по имени пользователя определяется, какие у него есть права доступа к объектам базы данных (подробнее это описывается в Главе 20). Следовательно, важно указать на этом этапе, к каким базам пользователь имеет право подключиться.

> **Примечание**
>
> Как можно узнать из Главы 20, Postgres Pro управляет правами и привилегиями, используя так называемые «роли». В этой главе под *пользователем* мы подразумеваем «роль с привилегией `LOGIN`».

Аутентификация это процесс идентификации клиента сервером базы данных, а также определение того, может ли клиентское приложение (или пользователь запустивший приложение) подключиться с указанным именем пользователя.

Postgres Pro предлагает несколько различных методов аутентификации клиентов. Метод аутентификации конкретного клиентского соединения может основываться на адресе компьютера клиента, имени базы данных, имени пользователя.

Имена пользователей базы данных Postgres Pro не имеют прямой связи с пользователями операционной системы на которой запущен сервер. Если у всех пользователей базы данных заведена учётная запись в операционной системе сервера, то имеет смысл назначить им точно такие же имена для входа в Postgres Pro. Однако сервер, принимающий удалённые подключения, может иметь большое количество пользователей базы данных, у которых нет учётной записи в ОС. В таких случаях не требуется соответствие между именами пользователей базы данных и именами пользователей операционной системы.

19.1. Файл `pg_hba.conf`

Аутентификация клиентов управляется конфигурационным файлом, который традиционно называется `pg_hba.conf` и расположен в каталоге с данными кластера базы данных. (HBA расшифровывается как host-based authentication — аутентификации по имени узла.) Файл `pg_hba.conf`, со стандартным содержимым, создаётся командой `initdb` при инициализации каталога с данными. Однако его можно разместить в любом другом месте; см. конфигурационный параметр hba_file.

Обычный формат файла `pg_hba.conf` представляет собой набор записей, по одной в строке. Пустые строки игнорируются, как и любой текст комментария после знака `#`. Записи не продолжаются на следующей строке. Записи состоят из некоторого количества полей, разделённых между собой пробелом и/или tabs. В полях могут быть использованы пробелы, если они взяты в кавычки. Если в кавычки берётся какое-либо зарезервированное слово в поле базы данных, пользователя или адресации (например, `all` или `replication`), то слово теряет своё особое значение и просто обозначает базу данных, пользователя или сервер с данным именем.

Каждая запись обозначает тип соединения, диапазон IP-адресов клиента (если он соотносится с типом соединения), имя базы данных, имя пользователя, и способ аутентификации, который будет использован для соединения в соответствии с этими параметрами. Первая запись с соответствующим типом соединения, адресом клиента, указанной базой данных и именем пользователя применяется для аутентификации. Процедур «fall-through» или «backup» не предусмотрено: если выбрана запись и аутентификация не прошла, последующие записи не рассматриваются. Если же ни одна из записей не подошла, в доступе будет отказано.

Запись может быть сделана в одном из семи форматов:

```
local      база  пользователь  метод-аутентификации  [параметры-аутентификации]
host       база  пользователь  адрес  метод-аутентификации  [параметры-аутентификации]
hostssl    база  пользователь  адрес  метод-аутентификации  [параметры-аутентификации]
hostnossl  база  пользователь  адрес  метод-аутентификации  [параметры-аутентификации]
host       база  пользователь  IP-адрес  IP-маска  метод-аутентификации  [параметры-
аутентификации]
hostssl    база  пользователь  IP-адрес  IP-маска  метод-аутентификации  [параметры-
аутентификации]
hostnossl  база  пользователь  IP-адрес  IP-маска  метод-аутентификации  [параметры-
аутентификации]
```

Значения полей описаны ниже:

`local`

Управляет подключениями через Unix-сокеты. Без подобной записи подключения через Unix-сокеты невозможны.

`host`

Управляет подключениями, устанавливаемыми по TCP/IP. Записи `host` соответствуют подключениям с SSL и без SSL.

> **Примечание**
>
> Удалённое соединение по TCP/IP невозможно, если сервер запущен без определения соответствующих значений для параметра конфигурации listen_addresses, поскольку по умолчанию система принимает подключения по TCP/IP только для локального адреса замыкания `localhost`.

`hostssl`

Управляет подключениями, устанавливаемыми по TCP/IP с применением шифрования SSL.

Чтобы использовать эту возможность, сервер должен быть собран с поддержкой SSL. Более того, механизм SSL должен быть включён параметром конфигурации ssl (подробнее об этом в Разделе 17.9). В противном случае запись `hostssl` не играет роли (не считая предупреждения о том, что ей не будут соответствовать никакие подключения).

`hostnossl`

Этот тип записей противоположен `hostssl`, ему соответствуют только подключения по TCP/IP без шифрования SSL.

база

Определяет, каким именам баз данных соответствует эта запись. Значение `all` определяет, что подходят все базы данных. Значение `sameuser` определяет, что данная запись соответствует только, если имя запрашиваемой базы данных совпадает с именем запрашиваемого пользователя. Значение `samerole` определяет, что запрашиваемый пользователь должен быть членом роли с таким же именем, как и у запрашиваемой базы данных. (`samegroup` — это устаревший, но допустимый вариант значения `samerole`.) Суперпользователи не становятся членами роли автоматически из-за `samerole`, а только если они являются явными членами роли, прямо или косвенно, и не только из-за того, что они суперпользователи. Значение `replication` показывает, что запись соответствует, если запрашивается подключение для физической репликации (имейте в виду, что для таких подключений не выбирается какая-то конкретная база данных). Любое другое значение воспринимается как имя определённой

базы Postgres Pro. Несколько имён баз данных можно указать, разделяя их запятыми. Файл, содержащий имена баз данных, можно указать, поставив знак `@` в начале его имени.

пользователь

Указывает, какому имени (или именам) пользователя базы данных соответствует эта запись. Значение `all` указывает, что запись соответствует всем пользователям. Любое другое значение задаёт либо имя конкретного пользователя базы данных, либо имя группы (если это значение начинается с `+`). (Напомним, что в Postgres Pro нет никакой разницы между пользователем и группой; знак `+` означает «совпадение любых ролей, которые прямо или косвенно являются членами роли», тогда как имя без знака `+` является подходящим только для этой конкретной роли.) В связи с этим, суперпользователь рассматривается как член роли, только если он явно является членом этой роли, прямо или косвенно, а не только потому, что он является суперпользователем. Несколько имён пользователей можно указать, разделяя их запятыми. Файл, содержащий имена пользователей, можно указать, поставив знак `@` в начале его имени.

адрес

Указывает адрес (или адреса) клиентской машины, которым соответствует данная запись. Это поле может содержать или имя компьютера, или диапазон IP-адресов, или одно из нижеупомянутых ключевых слов.

Диапазон IP-адресов указывается в виде начального адреса диапазона, дополненного косой чертой (`/`) и длиной маски CIDR. Длина маски задаёт количество старших битов клиентского IP-адреса, которые должны совпадать с битами IP-адреса диапазона. Биты, находящиеся правее, в указанном IP-адресе должны быть нулевыми. Между IP-адресом, знаком `/` и длиной маски CIDR не должно быть пробельных символов.

Типичные примеры диапазонов адресов IPv4, указанных таким образом: `172.20.143.89/32` для одного компьютера, `172.20.143.0/24` для небольшой и `10.6.0.0/16` для крупной сети. Диапазон адресов IPv6 может выглядеть как `::1/128` для одного компьютера (это адрес замыкания IPv6) или как `fe80::7a31:c1ff:0000:0000/96` для небольшой сети. `0.0.0.0/0` представляет все адреса IPv4, а `::0/0` — все адреса IPv6. Чтобы указать один компьютер, используйте длину маски 32 для IPv4 или 128 для IPv6. Опускать замыкающие нули в сетевом адресе нельзя.

Запись, сделанная в формате IPv4, подойдёт только для подключений по IPv4, а запись в формате IPv6 подойдёт только для подключений по IPv6, даже если представленный адрес находится в диапазоне IPv4-в-IPv6. Имейте в виду, что записи в формате IPv6 не будут приниматься, если системная библиотека C не поддерживает адреса IPv6.

Вы также можете прописать значение `all`, чтобы указать любой IP-адрес, `samehost`, чтобы указать любые IP-адреса данного сервера, или `samenet`, чтобы указать любой адрес любой подсети, к которой сервер подключён напрямую.

Если определено имя компьютера (всё, что не является диапазоном IP-адресов или специальным ключевым словом, воспринимается как имя компьютера), то оно сравнивается с результатом обратного преобразования IP-адреса клиента (например, обратного DNS-запроса, если используется DNS). При сравнении имён компьютеров регистр не учитывается. Если имена совпали, выполняется прямое преобразование имени (например, прямой DNS-запрос) для проверки, относится ли клиентский IP-адрес к адресам, соответствующим имени. Если двусторонняя проверка пройдена, запись считается соответствующей компьютеру. (В качестве имени узла в файле `pg_hba.conf` должно указываться то, что возвращается при преобразовании IP-адреса клиента в имя, иначе строка не будет соответствовать узлу. Некоторые базы данных имён позволяют связать с одним IP-адресом несколько имён узлов, но операционная система при попытке разрешить IP-адрес возвращает только одно имя.)

Указание имени, начинающееся с точки (`.`), соответствует суффиксу актуального имени узла. Так, `.example.com` будет соответствовать `foo.example.com` (а не только `example.com`).

Когда в pg_hba.conf указываются имена узлов, следует добиться, чтобы разрешение имён выполнялось достаточно быстро. Для этого может быть полезен локальный кеш разрешения имён, например, nscd. Вы также можете включить конфигурационный параметр log_hostname, чтобы видеть в журналах имя компьютера клиента вместо IP-адреса.

Это поле применимо только к записям host, hostssl и hostnossl.

> **Примечание**
>
> Пользователи часто задаются вопросом, почему имена серверов обрабатываются таким сложным, на первый взгляд, способом, с разрешением двух имён, включая обратный запрос клиентского IP-адреса. Это усложняет процесс в случае, если обратная DNS-запись клиента не установлена или включает в себя нежелательное имя узла. Такой способ избран, в первую очередь, для повышения эффективности: в этом случае соединение требует максимум два запроса разрешения, один прямой и один обратный. Если есть проблема разрешения с каким-то адресом, то она остаётся проблемой этого клиента. Гипотетически, могла бы быть реализована возможность во время каждой попытки соединения выполнять только прямой запрос для разрешения каждого имени сервера, упомянутого в pg_hba.conf. Но если список имён велик, процесс был бы довольно медленным, а в случае наличия проблемы разрешения у одного имени сервера, это стало бы общей проблемой.
>
> Также обратный запрос необходим для того, чтобы реализовать возможность соответствия суффиксов, поскольку для сопоставления с шаблоном требуется знать фактическое имя компьютера клиента.
>
> Обратите внимание, что такое поведение согласуется с другими популярными реализациями контроля доступа на основе имён, такими как Apache HTTP Server и TCP Wrappers.

IP-адрес
IP-маска

Эти два поля могут быть использованы как альтернатива записи *IP-адрес/длина-маски*. Вместо того, чтобы указывать длину маски, в отдельном столбце указывается сама маска. Например, 255.0.0.0 представляет собой маску CIDR для IPv4 длиной 8 бит, а 255.255.255.255 представляет маску CIDR длиной 32 бита.

Эти поля применимы только к записям host, hostssl и hostnossl.

метод-аутентификации

Указывает метод аутентификации, когда подключение соответствует этой записи. Варианты выбора приводятся ниже; подробности в Разделе 19.3. Все значения воспринимаются с учётом регистра и должны быть записаны в нижнем регистре, даже аббревиатуры типа ldap.

trust

Разрешает безусловное подключение. Этот метод позволяет тому, кто может подключиться к серверу с базой данных Postgres Pro, войти под любым желаемым пользователем Postgres Pro без введения пароля и без какой-либо другой аутентификации. За подробностями обратитесь к Подразделу 19.3.1.

reject

Отклоняет подключение безусловно. Эта возможность полезна для «фильтрации» некоторых серверов группы, например, строка reject может отклонить попытку подключения одного компьютера, при этом следующая строка позволяет подключиться остальным компьютерам в той же сети.

`scram-sha-256`

Проверяет пароль пользователя, производя аутентификацию SCRAM-SHA-256. За подробностями обратитесь к Подразделу 19.3.2.

`md5`

Проверяет пароль пользователя, производя аутентификацию SCRAM-SHA-256 или MD5. За подробностями обратитесь к Подразделу 19.3.2.

`password`

Требует для аутентификации введения клиентом незашифрованного пароля. Поскольку пароль посылается простым текстом через сеть, такой способ не стоит использовать, если сеть не вызывает доверия. За подробностями обратитесь к Подразделу 19.3.2.

`gss`

Для аутентификации пользователя использует GSSAPI. Этот способ доступен только для подключений по TCP/IP. За подробностями обратитесь к Подразделу 19.3.3.

`sspi`

Для аутентификации пользователя использует SSPI. Способ доступен только для Windows. За подробностями обратитесь к Подразделу 19.3.4.

`ident`

Получает имя пользователя операционной системы клиента, связываясь с сервером Ident, и проверяет, соответствует ли оно имени пользователя базы данных. Аутентификация ident может использоваться только для подключений по TCP/IP. Для локальных подключений применяется аутентификация peer. За подробностями обратитесь к Подразделу 19.3.5.

`peer`

Получает имя пользователя операционной системы клиента из операционной системы и проверяет, соответствует ли оно имени пользователя запрашиваемой базы данных. Доступно только для локальных подключений. За подробностями обратитесь к Подразделу 19.3.6.

`ldap`

Проводит аутентификацию, используя сервер LDAP. За подробностями обратитесь к Подразделу 19.3.7.

`radius`

Проводит аутентификацию, используя сервер RADIUS. За подробностями обратитесь к Подразделу 19.3.8

`cert`

Проводит аутентификацию, используя клиентский сертификат SSL. За подробностями обратитесь к Подразделу 19.3.9

`pam`

Проводит аутентификацию, используя службу подключаемых модулей аутентификации (PAM), предоставляемую операционной системой. За подробностями обратитесь к Подразделу 19.3.10.

`bsd`

Проводит аутентификацию, используя службу аутентификации BSD, предоставляемую операционной системой. За подробностями обратитесь к Подразделу 19.3.11.

.

параметры-аутентификации

После поля *метод-аутентификации* может идти поле (поля) вида *имя=значение*, определяющее параметры метода аутентификации. Подробнее о параметрах, доступных для различных методов аутентификации, рассказывается ниже.

Помимо описанных далее параметров, относящихся к различным методам, есть один общий параметр аутентификации `clientcert`, который можно задать в любой записи `hostssl`. Если он равен `1`, клиент должен представить подходящий (доверенный) сертификат SSL, в дополнение к другим требованиям метода проверки подлинности.

Файлы, включённые в конструкции, начинающиеся с `@`, читаются, как список имён, разделённых запятыми или пробелами. Комментарии предваряются знаком `#`, как и в файле `pg_hba.conf`, и вложенные `@` конструкции допустимы. Если только имя файла, начинающегося с `@` не является абсолютным путём.

Поскольку записи файла `pg_hba.conf` рассматриваются последовательно для каждого подключения, порядок записей имеет большое значение. Обычно более ранние записи определяют чёткие критерии для соответствия параметров подключения, но для методов аутентификации допускают послабления. Напротив, записи более поздние смягчают требования к соответствию параметров подключения, но усиливают их в отношении методов аутентификации. Например, некто желает использовать `trust` аутентификацию для локального подключения по TCP/IP, но при этом запрашивать пароль для удалённых подключений по TCP/IP. В этом случае запись, устанавливающая аутентификацию `trust` для подключения адреса 127.0.0.1, должна предшествовать записи, определяющей аутентификацию по паролю для более широкого диапазона клиентских IP-адресов.

Файл `pg_hba.conf` прочитывается при запуске системы, а также в тот момент, когда основной сервер получает сигнал SIGHUP. Если вы редактируете файл во время работы системы, необходимо послать сигнал процессу postmaster (используя `pg_ctl reload`, вызвав SQL-функцию `pg_reload_conf()` или выполнив `kill -HUP`), чтобы он прочел обновлённый файл.

Примечание

Предыдущее утверждение не касается Microsoft Windows: там любые изменения в `pg_hba.conf` сразу применяются к последующим подключениям.

Системное представление `pg_hba_file_rules` может быть полезно для предварительной проверки изменений в файле `pg_hba.conf` или для диагностики проблем, когда перезагрузка этого файла не даёт желаемого эффекта. Строки в этом представлении, содержащие в поле `error` не NULL, указывают на проблемы в соответствующих строках файла.

Подсказка

Чтобы подключиться к конкретной базе данных, пользователь не только должен пройти все проверки файла `pg_hba.conf`, но должен иметь привилегию `CONNECT` для подключения к базе данных. Если вы хотите ограничить доступ к базам данных для определённых пользователей, проще предоставить/отозвать привилегию `CONNECT`, нежели устанавливать правила в записях файла `pg_hba.conf`.

Примеры записей файла `pg_hba.conf` показаны в Примере 19.1. Обратитесь к следующему разделу за более подробной информацией по методам аутентификации.

Пример 19.1. Примеры записей `pg_hba.conf`

```
# Позволяет любому пользователю локальной системы подключаться
# к любой базе данных, используя любое имя пользователя баз данных, через
# Unix-сокеты (по умолчанию для локальных подключений).
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             all                                     trust

# То же, но для локальных замкнутых подключений по TCP/IP.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             127.0.0.1/32            trust

# То же, что и в предыдущей строке, но с указанием
# сетевой маски в отдельном столбце
#
# TYPE  DATABASE        USER            IP-ADDRESS      IP-MASK             METHOD
host    all             all             127.0.0.1       255.255.255.255     trust

# То же для IPv6.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             ::1/128                 trust

# То же самое, но с использованием имени компьютера
# (обычно покрывает и IPv4, и IPv6).
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             localhost               trust

# Позволяет любому пользователю любого компьютера с IP-адресом
# 192.168.93.x подключаться к базе данных "postgres"
# с именем, которое сообщает для данного подключения ident
# (как правило, имя пользователя операционной системы).
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    postgres        all             192.168.93.0/24         ident

# Позволяет любому пользователю компьютера 192.168.12.10 подключаться
# к базе данных "postgres", если он передаёт правильный пароль.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    postgres        all             192.168.12.10/32        scram-sha-256

# Позволяет любым пользователям с компьютеров в домене example.com
# подключаться к любой базе данных, если передаётся правильный пароль.
#
# Для всех пользователей требуется аутентификация SCRAM, за исключением
# пользователя 'mike', который использует старый клиент, не поддерживающий
# аутентификацию SCRAM.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             mike            .example.com            md5
host    all             all             .example.com            scram-sha-256

# В случае отсутствия предшествующих строчек с "host", следующие две строки
# откажут в подключении с 192.168.54.1 (поскольку данная запись будет
```

```
# выбрана первой), но разрешат подключения GSSAPI с любых других
# адресов. С нулевой маской ни один бит из IP-адреса компьютера
# не учитывается, так что этой строке соответствует любой компьютер.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             192.168.54.1/32         reject
host    all             all             0.0.0.0/0               gss

# Позволяет пользователям с любого компьютера 192.168.x.x подключаться
# к любой базе данных, если они проходят проверку ident.  Если же ident
# говорит, например, что это пользователь "bryanh" и он запрашивает
# подключение как пользователь Postgres Pro "guest1", подключение
# будет разрешено, если в файле pg_ident.conf есть сопоставление
# "omicron", позволяющее пользователю "bryanh" подключаться как "guest1".
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             192.168.0.0/16          ident map=omicron

# Если для локальных подключений предусмотрены только эти три строки,
# они позволят локальным пользователям подключаться только к своим
# базам данных (базам данных с именами, совпадающими с
# именами пользователей баз данных), кроме администраторов
# или членов роли "support", которые могут подключиться к любой БД.
# Список имён администраторов содержится в файле $PGDATA/admins.
# Пароли запрашиваются в любом случае.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   sameuser        all                                     md5
local   all             @admins                                 md5
local   all             +support                                md5

# Последние две строчки выше могут быть объединены в одну:
local   all             @admins,+support                        md5

# В столбце DATABASE также могут указываться списки и имена файлов:
local   db1,db2,@demodbs  all                                   md5
```

19.2. Файл сопоставления имён пользователей

Когда используется внешняя система аутентификации, например Ident или GSSAPI, имя пользователя операционной системы, устанавливающего подключение, может не совпадать с именем целевого пользователя (роли) базы данных. В этом случае можно применить сопоставление имён пользователей, чтобы сменить имя пользователя операционной системы на имя пользователя БД. Чтобы задействовать сопоставление имён, укажите `map=`*`имя-сопоставления`* в поле параметров в `pg_hba.conf`. Этот параметр поддерживается для всех методов аутентификации, которые принимают внешние имена пользователей. Так как для разных подключений могут требоваться разные сопоставления, сопоставление определяется параметром *`имя-сопоставления`* в `pg_hba.conf` для каждого отдельного подключения.

Сопоставления имён пользователя определяются в файле сопоставления ident, который по умолчанию называется `pg_ident.conf`и хранится в каталоге данных кластера. (Файл сопоставления может быть помещён и в другое место, обратитесь к информации о настройке параметра ident_file.) Файл сопоставления ident содержит строки общей формы:

`map-name system-username database-username`

Комментарии и пробелы применяются так же, как и в файле `pg_hba.conf`. *`map-name`* является произвольным именем, на которое будет ссылаться файл сопоставления файла `pg_hba.conf`. Два других поля указывают имя пользователя операционной системы и соответствующее имя

пользователя базы данных. Имя *map-name* может быть использовано неоднократно, чтобы указывать множественные сопоставления пользовательских имён в рамках одного файла сопоставления.

Нет никаких ограничений по количеству пользователей баз данных, на которые может ссылаться пользователь операционной системы, и наоборот. Тем не менее записи в файле скорее подразумевают, что « пользователь этой операционной системы может подключиться как пользователь этой базы данных», нежели показывают, что эти имена пользователей эквивалентны. Подключение разрешается, если существует запись в файле сопоставления, соединяющая имя, полученное от внешней системы аутентификации, с именем пользователя базы данных, под которым пользователь хочет подключиться.

Если поле *system-username* начинается со знака (`/`), оставшаяся его часть рассматривается как регулярное выражение. (Подробнее синтаксис регулярных выражений Postgres Pro описан в Подразделе 9.7.3.1.) Регулярное выражение может включать в себя одну группу, или заключённое в скобки подвыражение, на которое можно сослаться в поле *database-username*, написав `\1` (с одной обратной косой). Это позволяет сопоставить несколько имён пользователя с одной строкой, что особенно удобно для простых замен. Например, эти строки

```
mymap    /^(.*)@mydomain\.com$       \1
mymap    /^(.*)@otherdomain\.com$    guest
```

удалят часть домена для имён пользователей, которые заканчиваются на `@mydomain.com`, и позволят пользователям, чьё имя пользователя системы заканчивается на `@otherdomain.com`, подключиться как `guest`.

> **Подсказка**
>
> Помните, что по умолчанию, регулярное выражение может совпасть только с частью строки. Разумным выходом будет использование символов `^` и `$`, как показано в примере выше, для принудительного совпадения со всем именем пользователя операционной системы

Файл `pg_ident.conf` прочитывается при запуске системы, а также в тот момент, когда основной сервер получает сигнал SIGHUP. Если вы редактируете файл во время работы системы, необходимо послать сигнал процессу postmaster (используя `pg_ctl reload`, вызвав SQL-функцию `pg_reload_conf()` или выполнив `kill -HUP`), чтобы он прочел обновлённый файл.

Файл `pg_ident.conf`, который может быть использован в сочетании с файлом `pg_hba.conf` (см. Пример 19.1), показан в Примере 19.2. В этом примере любым пользователям компьютеров в сети 192.168 с именами, отличными от `bryanh`, `ann` или `robert`, будет отказано в доступе. Пользователь системы `robert` получит доступ только тогда, когда подключается как пользователь Postgres Pro `bob`, а не как `robert`, или какой-либо другой пользователь. Пользователь `ann` сможет подключиться только как `ann`. Пользователь `bryanh` сможет подключиться как `bryanh` или как `guest1`.

Пример 19.2. Пример файла `pg_ident.conf`

```
# MAPNAME       SYSTEM-USERNAME         PG-USERNAME

omicron         bryanh                  bryanh
omicron         ann                     ann
# на этих машинах у пользователя bob имя robert
omicron         robert                  bob
# bryanh также может подключаться как guest1
omicron         bryanh                  guest1
```

19.3. Методы аутентификации

Следующие подразделы содержат более детальную информацию о методах аутентификации.

19.3.1. Аутентификация trust

Когда указан способ аутентификации `trust`, Postgres Pro предполагает, что любой подключающийся к серверу авторизован для доступа к базе данных вне зависимости от указанного имени пользователя базы данных (даже если это имя суперпользователя). Конечно, ограничения, прописанные в столбцах `база` и `пользователь`, продолжают работать. Этот метод должен применяться только в том случае, когда на уровне операционной системы обеспечена адекватная защита от подключений к серверу.

Аутентификация `trust` очень удобна для локальных подключений на однопользовательской рабочей станции. Но сам по себе этот метод обычно *не* подходит для машин с несколькими пользователями. Однако вы можете использовать `trust` даже на многопользовательской машине, если ограничите доступ к файлу Unix-сокета сервера на уровне файловой системы. Для этого установите конфигурационные параметры `unix_socket_permissions` (и, возможно, `unix_socket_group`) как описано в Разделе 18.3. Либо вы можете установить конфигурационный параметр `unix_socket_directories`, чтобы разместить файл сокета в должным образом защищённом каталоге.

Установка разрешений на уровне файловой системы помогает только в случае подключений через Unix-сокеты. На локальные подключения по TCP/IP ограничения файловой системы не влияют. Поэтому, если вы хотите использовать разрешения файловой системы для обеспечения локальной безопасности, уберите строку `host ... 127.0.0.1 ...` из `pg_hba.conf` или смените метод аутентификации.

Метод аутентификации `trust` для подключений по TCP/IP допустим только в случае, если вы доверяете каждому пользователю компьютера, получившему разрешение на подключение к серверу строками файла `pg_hba.conf`, указывающими метод `trust`. Не стоит использовать `trust` для любых подключений по TCP/IP, отличных от localhost (127.0.0.1).

19.3.2. Аутентификация password

Существует несколько методов аутентификации по паролю. Они работают примерно одинаково, но различаются тем, как пароли пользователей хранятся на сервере и как пароль передаётся от клиента по каналу связи.

`scram-sha-256`

С методом `scram-sha-256` выполняется аутентификация SCRAM-SHA-256, как описано в *RFC 7677*. Она производится по схеме вызов-ответ, которая предотвращает перехват паролей через недоверенные соединения и поддерживает хранение паролей на сервере в виде криптографического хеша, что считается безопасным.

Это наиболее безопасный из существующих на данный момент методов, но он не поддерживается старыми клиентскими библиотеками.

`md5`

Для метода `md5` реализован менее безопасный механизм вызов-ответ. Он предотвращает перехват паролей и предусматривает хранение паролей на сервере в зашифрованном виде, но не защищает в случае похищения хешей паролей с сервера. Кроме того, алгоритм хеширования MD5 в наши дни уже может не защитить от целенаправленных атак.

Метод `md5` несовместим с функциональностью db_user_namespace.

Для облегчения перехода от метода `md5` к более новому методу SCRAM, если в качестве метода аутентификации в `pg_hba.conf` указан `md5`, но пароль пользователя на сервере зашифрован для SCRAM (см. ниже), автоматически будет производиться аутентификация на базе SCRAM.

`password`

С методом `password` пароль передаётся в открытом виде и поэтому является уязвимым для атак с перехватом трафика. Его следует избегать всегда, если это возможно. Однако если

подключение защищено SSL, метод password может быть безопасен. (Хотя аутентификация по сертификату SSL может быть лучшим выбором когда используется SSL).

Пароли баз данных Postgres Pro отделены от паролей пользователей операционной системы. Пароли всех пользователей базы данных хранятся во внутреннем каталоге pg_authid. Управлять паролями можно, либо используя SQL-команды CREATE USER и ALTER ROLE, например, **CREATE USER foo WITH PASSWORD 'secret'**, либо с помощью команды psql \password. Если пароль для пользователя не задан, вместо него хранится NULL, и пройти аутентификацию по паролю этот пользователь не сможет.

Доступность различных методов аутентификации по паролю зависит от того, как пароли пользователей шифруются на сервере (или, говоря точнее, хешируются). Это определяется параметром конфигурации password_encryption в момент назначения пароля. Если пароль шифруется в режиме scram-sha-256, его можно будет использовать для методов аутентификации scram-sha-256 и password (но в последнем случае он будет передаваться открытым текстом). В случае указания метода аутентификации md5 при этом произойдёт автоматический переход к использованию scram-sha-256, как сказано выше, так что этот вариант тоже будет работать. Если пароль шифруется в режиме md5, его можно будет использовать только для методов аутентификации md5 и password (и в последнем случае он так же будет передаваться открытым текстом). (Ранние версии Postgres Pro поддерживали хранение паролей на сервере в открытом виде, но теперь это невозможно.) Чтобы просмотреть хранящиеся в БД хеши паролей, обратитесь к системному каталогу pg_authid.

Для перевода существующей инсталляции с md5 на scram-sha-256, после того как все клиентские библиотеки будут обновлены до версий, поддерживающих SCRAM, задайте password_encryption = 'scram-sha-256' в postgresql.conf, добейтесь, чтобы все пользователи сменили свои пароли, а затем поменяйте указания метода аутентификации в pg_hba.conf на scram-sha-256.

19.3.3. Аутентификация GSSAPI

GSSAPI является протоколом отраслевого стандарта для безопасной авторизации, определённым в RFC 2743. Postgres Pro поддерживает GSSAPI с Kerberos аутентификацией с соответствии с RFC 1964. GSSAPI обеспечивает автоматическую аутентификацию (single sign-on), для систем, которые её поддерживают. Сама по себе аутентификация безопасна, но данные, отсылаемые в ходе подключения к базе данных, не защищены, если не используется SSL.

Поддержка GSSAPI должна быть включена при сборке Postgres Pro Standard.

При работе с Kerberos GSSAPI использует стандартные учётные записи в формате *servicename*/*hostname*@*realm*. Сервер Postgres Pro примет любого принципала, включённого в используемый сервером файл таблицы ключей, но необходимо проявить осторожность в указании корректных деталей принципала в ходе соединения с клиентом, применяющим параметр подключения krbsrvname. (См. также Подраздел 31.1.2.) Значение имени сервиса по умолчанию postgres может быть изменено во время сборки с помощью ./configure --with-krb-srvnam=*whatever*. В большинстве сред изменять данный параметр не требуется. Однако некоторые реализации Kerberos могут потребовать иного имени сервиса, например, Microsoft Active Directory требует, чтобы имя сервиса было набрано заглавными буквами (POSTGRES).

hostname здесь — это полное доменное имя компьютера, где работает сервер. Областью субъекта-службы является предпочитаемая область данного компьютера.

Принципалы клиентов могут быть сопоставлены с различными именами пользователей баз данных Postgres Pro в pg_ident.conf. Например, принципалу pgusername@realm может быть сопоставлено просто pgusername. Так же возможно использовать в качестве имени роли в Postgres Pro полное имя принципала username@realm без какого-либо сопоставления.

Postgres Pro также поддерживает возможность убирать область из имени принципала. Эта возможность оставлена для обратной совместимости и использовать её крайне нежелательно, так как при этом оказывается невозможно различить разных пользователей, имеющих одинаковые

имена, но приходящих из разных областей. Чтобы включить её, установите для include_realm значение 0. В простых конфигурациях с одной областью исключение области в сочетании с параметром krb_realm (который позволяет ограничить область пользователя одним значением, заданным в krb_realm parameter) будет безопасным, но менее гибким вариантом по сравнению с явным описанием сопоставлений в pg_ident.conf.

Убедитесь, что файл ключей вашего сервера доступен для чтения (и желательно недоступен для записи) учётной записи сервера Postgres Pro. (См. также Раздел 17.1.) Расположение этого файла ключей указывается параметром krb_server_keyfile. По умолчанию это /usr/local/pgsql/etc/krb5.keytab (каталог может быть другим, в зависимости от значения sysconfdir при сборке). Из соображений безопасности рекомендуется использовать отдельный файл keytab для сервера Postgres Pro, а не открывать доступ к общесистемному файлу.

Файл таблицы ключей генерируется программным обеспечением Kerberos; подробнее это описано в документации Kerberos. Следующий пример для MIT-совместимых реализаций Kerberos 5:

```
kadmin% ank -randkey postgres/server.my.domain.org
kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

При подключении к базе данных убедитесь, что у вас есть разрешение на сопоставление принципала с именем пользователя базы данных. Например, для имени пользователя базы данных fred, принципал fred@EXAMPLE.COM сможет подключиться. Чтобы дать разрешение на подключение принципалу fred/users.example.com@EXAMPLE.COM, используйте файл сопоставления имён пользователей, как описано в Разделе 19.2.

Для метода GSSAPI доступны следующие параметры конфигурации:

include_realm

Когда этот параметр равен 0, из принципала аутентифицированного пользователя убирается область, и оставшееся имя проходит сопоставление имён (см. Раздел 19.2). Этот вариант не рекомендуется и поддерживается в основном для обратной совместимости, так как он небезопасен в окружениях с несколькими областями, если только дополнительно не задаётся krb_realm. Более предпочтительный вариант — оставить значение include_realm по умолчанию (1) и задать в pg_ident.conf явное сопоставление для преобразования имён принципалов в имена пользователей Postgres Pro.

map

Разрешает сопоставление имён пользователей системы и пользователей баз данных. За подробностями обратитесь к Разделу 19.2. Для принципала GSSAPI/Kerberos, такого как username@EXAMPLE.COM (или более редкого username/hostbased@EXAMPLE.COM), именем пользователя в сопоставлении будет username@EXAMPLE.COM (или username/hostbased@EXAMPLE.COM, соответственно), если include_realm не равно 0; в противном случае именем системного пользователя в сопоставлении будет username (или username/hostbased).

krb_realm

Устанавливает область, с которой будут сверяться имена принципалов пользователей. Если этот параметр задан, подключаться смогут только пользователи из этой области. Если не задан, подключаться смогут пользователи из любой области, в зависимости от установленного сопоставления имён пользователей.

19.3.4. Аутентификация SSPI

SSPI — технология Windows для защищённой аутентификации с единственным входом. Postgres Pro использует SSPI в режиме negotiate, который применяет Kerberos, когда это возможно, и автоматически возвращается к NTLM в других случаях. Аутентификация SSPI возможна только когда и сервер, и клиент работают на платформе Windows или на других платформах, где доступен GSSAPI.

Если используется аутентификация Kerberos, SSPI работает так же, как GSSAPI; подробнее об этом рассказывается в Подразделе 19.3.3.

Для SSPI доступны следующие параметры конфигурации:

`include_realm`

Когда этот параметр равен 0, из принципала аутентифицированного пользователя убирается область, и оставшееся имя проходит сопоставление имён (см. Раздел 19.2). Этот вариант не рекомендуется и поддерживается в основном для обратной совместимости, так как он небезопасен в окружениях с несколькими областями, если только дополнительно не задаётся `krb_realm`. Более предпочтительный вариант — оставить значение `include_realm` по умолчанию (1) и задать в `pg_ident.conf` явное сопоставление для преобразования имён принципалов в имена пользователей Postgres Pro.

`compat_realm`

Если равен 1, для параметра `include_realm` применяется имя домена, совместимое с SAM (также известное как имя NetBIOS). Это вариант по умолчанию. Если он равен 0, для имени принципала Kerberos применяется действительное имя области.

Этот параметр можно отключить, только если ваш сервер работает под именем доменного пользователя (в том числе, виртуального пользователя службы на компьютере, включённом в домен) и все клиенты, проходящие проверку подлинности через SSPI, также используют доменные учётные записи; в противном случае аутентификация не будет выполнена.

`upn_username`

Если этот параметр включён вместе с `compat_realm`, для аутентификации применяется имя Kerberos UPN. Если он отключён (по умолчанию), применяется SAM-совместимое имя пользователя. По умолчанию у новых учётных записей эти два имени совпадают.

Заметьте, что libpq использует имя, совместимое с SAM, если имя не задано явно. Если вы применяете libpq или драйвер на его базе, этот параметр следует оставить отключённым, либо явно задавать имя пользователя в строке подключения.

`map`

Позволяет сопоставить пользователей системы с пользователями баз данных. За подробностями обратитесь к Разделу 19.2. Для принципала SSPI/Kerberos, такого как `username@EXAMPLE.COM` (или более редкого `username/hostbased@EXAMPLE.COM`), именем пользователя в сопоставлении будет `username@EXAMPLE.COM` (или `username/hostbased@EXAMPLE.COM`, соответственно), если `include_realm` не равно 0; в противном случае именем системного пользователя в сопоставлении будет `username` (или `username/hostbased`).

`krb_realm`

Устанавливает область, с которой будут сверяться имена принципалов пользователей. Если этот параметр задан, подключаться смогут только пользователи из этой области. Если не задан, подключаться смогут пользователи из любой области, в зависимости от установленного сопоставления имён пользователей.

19.3.5. Аутентификация ident

Метод аутентификации ident работает, получая имя пользователя операционной системы клиента от сервера Ident и используя его в качестве разрешённого имени пользователя базы данных (с возможным сопоставлением имён пользователя). Способ доступен только для подключений по TCP/IP.

> **Примечание**
>
> Когда для локального подключения (не TCP/IP) указан ident, вместо него используется метод аутентификации peer (см. Подраздел 19.3.6).

Для метода ident доступны следующие параметры конфигурации:

`map`

Позволяет сопоставить имена пользователей системы и базы данных. За подробностями обратитесь к Разделу 19.2.

Протокол «Identification» (Ident) описан в RFC 1413. Практически каждая Unix-подобная операционная система поставляется с сервером Ident, по умолчанию слушающим TCP-порт 113. Базовая функция этого сервера — отвечать на вопросы, вроде «Какой пользователь инициировал подключение, которое идет через твой порт *X* и подключается к моему порту *Y*?». Поскольку после установления физического подключения Postgres Pro знает и *X*, и *Y*, он может опрашивать сервер Ident на компьютере клиента и теоретически может определять пользователя операционной системы при каждом подключении.

Недостатком этой процедуры является то, что она зависит от интеграции с клиентом: если клиентская машина не вызывает доверия или скомпрометирована, злоумышленник может запустить любую программу на порту 113 и вернуть любое имя пользователя на свой выбор. Поэтому этот метод аутентификации подходит только для закрытых сетей, где каждая клиентская машина находится под жёстким контролем и где администраторы операционных систем и баз данных работают в тесном контакте. Другими словами, вы должны доверять машине, на которой работает сервер Ident. Помните предупреждение:

> Протокол Ident не предназначен для использования в качестве протокола авторизации и контроля доступа.
>
> —RFC 1413

У некоторых серверов Ident есть нестандартная возможность, позволяющая зашифровать возвращаемое имя пользователя, используя ключ, который известен только администратору исходного компьютера. Эту возможность *нельзя* использовать с Postgres Pro, поскольку Postgres Pro не сможет расшифровать возвращаемую строку и получить фактическое имя пользователя.

19.3.6. Аутентификация peer

Метод аутентификации peer работает, получая имя пользователя операционной системы клиента из ядра и используя его в качестве разрешённого имени пользователя базы данных (с возможностью сопоставления имён пользователя). Этот метод поддерживается только для локальных подключений.

Для метода peer доступны следующие параметры конфигурации:

`map`

Позволяет сопоставить имена пользователей системы и базы данных. За подробностями обратитесь к Разделу 19.2.

Аутентификация peer доступна только в операционных системах, поддерживающих функцию `getpeereid()`, параметр сокета `SO_PEERCRED` или подобные механизмы. В настоящее время это Linux, большая часть разновидностей BSD, включая macOS, и Solaris.

19.3.7. Аутентификация LDAP

Данный метод аутентификации работает сходным с методом `password` образом, за исключением того, что он использует LDAP как метод подтверждения пароля. LDAP используется только для подтверждения пары «имя пользователя/пароль». Поэтому пользователь должен уже существовать в базе данных до того, как для аутентификации будет использован LDAP.

Аутентификация LDAP может работать в двух режимах. Первый режим называется простое связывание. В ходе аутентификации сервер связывается с характерным именем, составленным следующим образом: *`prefix username suffix`*. Обычно, параметр *`prefix`* используется для указания `cn=` или *`DOMAIN\`* в среде Active Directory. *`suffix`* используется для указания оставшейся части DN или в среде, отличной от Active Directory.

Во втором режиме, который мы называем поиск+связывание, сервер сначала связывается с каталогом LDAP с предопределённым именем пользователя и паролем, указанным в *ldapbinddn* и *ldapbindpasswd*, и выполняет поиск пользователя, пытающегося подключиться к базе данных. Если имя пользователя и пароль не определены, сервер пытается связаться с каталогом анонимно. Поиск выполняется в поддереве *ldapbasedn*, при этом проверятся точное соответствие имени пользователя атрибуту *ldapsearchattribute*. Как только при поиске находится пользователь, сервер отключается и заново связывается с каталогом уже как этот пользователь, с паролем, переданным клиентом, чтобы удостовериться, что учётная запись корректна. Этот же режим используется в схемах LDAP-аутентификации в другом программном обеспечении, например, в pam_ldap и mod_authnz_ldap в Apache. Данный вариант даёт больше гибкости в выборе расположения объектов пользователей, но при этом требует дважды подключаться к серверу LDAP.

Следующие параметры конфигурации доступны при аутентификации в обоих режимах:

`ldapserver`

Имена и IP-адреса LDAP-серверов для связи. Можно указать несколько серверов, разделяя их пробелами.

`ldapport`

Номер порта для связи с LDAP-сервером. Если порт не указан, используется установленный по умолчанию порт библиотеки LDAP.

`ldaptls`

Равен 1 для установки соединения между Postgres Pro и LDAP-сервером с использованием TLS-шифрования. Имейте в виду, что так шифруется только обмен данными с LDAP-сервером, а клиентское подключение остаётся незашифрованным, если только не применяется SSL.

Следующие параметры конфигурации доступны только при аутентификации в режиме простого связывания:

`ldapprefix`

Эта строка подставляется перед именем пользователя во время формирования DN для связывания при аутентификации в режиме простого связывания.

`ldapsuffix`

Эта строка размещается после имени пользователя во время формирования DN для связывания, при аутентификации в режиме простого связывания.

Следующие параметры конфигурации доступны только при аутентификации поиск+связывание:

`ldapbasedn`

Корневая папка DN для начала поиска пользователя при аутентификации в режиме поиск +связывание.

`ldapbinddn`

DN пользователя для связи с каталогом при выполнении поиска в ходе аутентификации в режиме поиск+связывание.

`ldapbindpasswd`

Пароль пользователя для связывания с каталогом при выполнении поиска в ходе аутентификации в режиме поиск+связывание.

`ldapsearchattribute`

Атрибут для соотнесения с именем пользователя в ходе аутентификации поиск+связывание. Если атрибут не указан, будет использован атрибут `uid`.

`ldapurl`

Адрес RFC 4516 LDAP. Это альтернативный путь для написания некоторых функций LDAP в более компактной и стандартной форме. Формат записи таков:

`ldap://`*`host`*`[:`*`port`*`]/`*`basedn`*`[?[`*`attribute`*`][?[`*`scope`*`]]]`

`scope` должен быть представлен или `base`, или `one`, или `sub`, обычно последним. Используется один атрибут, некоторые компоненты стандартных LDAP-адресов, такие, как фильтры и расширения, не поддерживаются.

Для неанонимного связывания `ldapbinddn` и `ldapbindpasswd` должны быть указаны как раздельные параметры.

Для применения зашифрованных LDAP-подключений, в дополнение к параметру `ldapurl` необходимо использовать параметр `ldaptls`. URL-схема `ldaps` (прямое SSL-подключение) не поддерживается.

В настоящее время URL-адреса LDAP поддерживаются только с OpenLDAP и не поддерживаются в Windows.

Нельзя путать параметры конфигурации для режима простого связывания с параметрами для режима поиск+связывание, это ошибка.

Это пример конфигурации LDAP для простого связывания:

```
host ... ldap ldapserver=ldap.example.net ldapprefix="cn=" ldapsuffix=", dc=example,
 dc=net"
```

Когда запрашивается подключение к серверу базы данных в качестве пользователя базы данных `someuser`, Postgres Pro пытается связаться с LDAP-сервером, используя DN `cn=someuser, dc=example, dc=net` и пароль, предоставленный клиентом. Если это подключение удалось, то доступ к базе данных будет открыт.

Пример конфигурации для режима поиск+связывание:

```
host ... ldap ldapserver=ldap.example.net ldapbasedn="dc=example, dc=net"
 ldapsearchattribute=uid
```

Когда запрашивается подключение к серверу базы данных в качестве пользователя базы данных `someuser`, Postgres Pro пытается связаться с сервером LDAP анонимно (поскольку `ldapbinddn` не был указан), выполняет поиск для `(uid=someuser)` под указанной базой DN. Если запись найдена, проводится попытка связывание с использованием найденной информации и паролем, предоставленным клиентом. Если вторая попытка подключения проходит успешно, предоставляется доступ к базе данных.

Пример той же конфигурации для режима поиск+связывание, но записанной в виде URL:

```
host ... ldap ldapurl="ldap://ldap.example.net/dc=example,dc=net?uid?sub"
```

Такой URL-формат используется и другим программным обеспечением, поддерживающим аутентификацию по протоколу LDAP, поэтому распространять такую конфигурацию будет легче.

> **Подсказка**
>
> Поскольку LDAP часто применяет запятые и пробелы для разделения различных частей DN, необходимо использовать кавычки при определении значения параметров, как показано в наших примерах.

19.3.8. Аутентификация RADIUS

Данный метод аутентификации работает сходным с методом `password` образом, за исключением того, что он использует RADIUS как метод проверки пароля. RADIUS используется только

для подтверждения пары «имя пользователя/пароль». Поэтому пользователь должен уже существовать в базе данных до того, как для аутентификации будет использован RADIUS.

В ходе аутентификации RADIUS настроенному RADIUS-серверу посылается запрос доступа. Это сообщение типа `Authenticate Only` (Только аутентификация), которое включает в себя параметры `user name` (имя пользователя), `password` (зашифрованный пароль) и `NAS Identifier` (идентификатор NAS). Запрос зашифровывается с использованием общего с сервером секрета. RADIUS-сервер отвечает на этот запрос либо `Access Accept` (Доступ принят), либо `Access Reject` (Доступ отклонён). Система ведения учёта RADIUS не поддерживается.

Для данного метода можно указать адреса нескольких серверов RADIUS, тогда они будут перебираться по очереди. В случае получения от любого сервера отрицательного ответа произойдёт сбой аутентификации. Если ответ не будет получен, последует попытка подключения к следующему серверу в списке. Чтобы задать имена нескольких серверов, разделите их имена запятыми и заключите список в двойные кавычки. При этом все остальные параметры RADIUS должны указываться так же в списках через запятую, чтобы каждый сервер получил собственное значение. Возможно также задавать их единственным значением, тогда это значение будет применяться ко всем серверам.

Для метода RADIUS доступны следующие параметры конфигурации:

`radiusservers`

DNS-имена или IP-адреса целевых серверов RADIUS. Это обязательный параметр.

`radiussecrets`

Общие секреты, используемые при контактах с серверами RADIUS. Значение этого параметра должно быть одинаковым на серверах Postgres Pro и RADIUS. Рекомендуется использовать строку как минимум из 16 символов. Это обязательный параметр.

> **Примечание**
>
> Шифровальный вектор будет достаточно эффективен только в том случае, если Postgres Pro собран с поддержкой OpenSSL. В противном случае передача данных серверу RADIUS будет лишь замаскированной, но не защищённой, поэтому необходимо принять дополнительные меры безопасности.

`radiusports`

Номера портов для подключения к серверам RADIUS. Если порты не указываются, используется стандартный порт RADIUS (`1812`).

`radiusidentifiers`

Строки, используемые в запросах RADIUS как `NAS Identifier` (Идентификатор NAS). Этот параметр может использоваться, например, для обозначения кластера БД, к которому пытается подключаться пользователь, что позволяет выбрать соответствующую политику на сервере RADIUS. Если никакой идентификатор не задан, по умолчанию используется `postgresql`.

Если в значении параметра RADIUS необходимо передать запятую или пробел, это можно сделать, заключив это значение в двойные кавычки, хотя это может быть не очень удобно, так как потребуются два уровня двойных кавычек. Например, так добавляются пробелы в строки секретов RADIUS:

```
host ... radius radiusservers="server1,server2" radiussecrets="""secret one"",""secret
 two"""
```

19.3.9. Аутентификация по сертификату

Для аутентификации в рамках этого метода используется клиентский сертификат SSL, поэтому данный способ применим только для SSL-подключений. Когда используется этот метод, сервер потребует от клиента предъявления действительного и доверенного сертификата. Пароль у клиента не запрашивается. Атрибут `cn` (Обычное имя) сертификата сравнивается с запрашиваемым именем пользователя базы данных, и если они соответствуют, вход разрешается. Если `cn` отличается от имени пользователя базы данных, то может быть использовано сопоставление имён пользователей.

Для аутентификации по SSL сертификату доступны следующие параметры конфигурации:

`map`

Позволяет сопоставить имена пользователей системы и базы данных. За подробностями обратитесь к Разделу 19.2.

В записи `pg_hba.conf`, описывающей аутентификацию по сертификату, параметр `clientcert` предполагается равным `1`, и его нельзя отключить, так как для этого метода клиентский сертификат является обязательным. Метод `cert` отличается от простой проверки пригодности сертификата `clientcert` только тем, что также проверяет, соответствует ли атрибут `cn` имени пользователя базы данных.

19.3.10. Аутентификация PAM

Данный метод аутентификации работает подобно методу `password`, но использует в качестве механизма проверки подлинности PAM (Pluggable Authentication Modules, Подключаемые модули аутентификации). По умолчанию имя службы PAM — `postgresql`. PAM используется только для проверки пар «имя пользователя/пароль» и может дополнительно проверять имя или IP-адрес удалённого компьютера. Поэтому пользователь должен уже существовать в базе данных, чтобы PAM можно было использовать для аутентификации. За дополнительной информацией о PAM обратитесь к *Странице описания Linux-PAM*.

Для аутентификации PAM доступны следующие параметры конфигурации:

`pamservice`

Имя службы PAM

`pam_use_hostname`

Указывает, предоставляется ли модулям PAM через поле `PAM_RHOST` IP-адрес либо имя удалённого компьютера. По умолчанию выдаётся IP-адрес. Установите в этом параметре 1, чтобы использовать имя узла. Разрешение имени узла может приводить к задержкам при подключении. (Обычно конфигурации PAM не задействуют эту информацию, так что этот параметр следует учитывать, только если создана специальная конфигурация, в которой он используется.)

> **Примечание**
>
> Если PAM настроен для чтения `/etc/shadow`, произойдёт сбой аутентификации, потому что сервер Postgres Pro запущен не пользователем root. Однако это не имеет значения, когда PAM настроен для использования LDAP или других методов аутентификации.

19.3.11. Аутентификация BSD

Данный метод аутентификации работает подобно методу `password`, но использует для проверки пароля механизм аутентификации BSD. Аутентификация BSD используется только для проверки пар «имя пользователя/пароль». Поэтому роль пользователя должна уже существовать в базе данных, чтобы эта аутентификация была успешной. Механизм аутентификации BSD в настоящее время может применяться только в OpenBSD.

Для аутентификации BSD в Postgres Pro применяется тип входа `auth-postgresql` и класс `postgresql`, если он определён в `login.conf`. По умолчанию этот класс входа не существует и Postgres Pro использует класс входа по умолчанию.

> **Примечание**
>
> Для использования аутентификации BSD необходимо сначала добавить учётную запись пользователя Postgres Pro (то есть, пользователя ОС, запускающего сервер) в группу `auth`. Группа `auth` существует в системах OpenBSD по умолчанию.

19.4. Проблемы аутентификации

Сбои и другие проблемы с аутентификацией обычно дают о себе знать через сообщения об ошибках, например:

```
FATAL:  no pg_hba.conf entry for host "123.123.123.123", user "andym", database
 "testdb"
```

Это сообщение вы, скорее всего, получите, если сможете связаться с сервером, но он не захочет с вами общаться. В сообщении содержится предположение, что сервер отказывает вам в подключении, поскольку не может найти подходящую запись в файле `pg_hba.conf`.

```
FATAL:  password authentication failed for user "andym"
```

Такое сообщение показывает, что вы связались с сервером, он готов общаться с вами, но только после того, как вы прошли авторизацию по методу, указанному в файле `pg_hba.conf`. Проверьте пароль, который вы вводите, и как настроен Kerberos или ident, если в сообщении упоминается один из этих типов аутентификации.

```
FATAL:  user "andym" does not exist
```

Указанное имя пользователя базы данных не найдено.

```
FATAL:  database "testdb" does not exist
```

База данных, к которой вы пытаетесь подключиться, не существует. Имейте в виду, что если вы не указали имя базы данных, по умолчанию берётся имя пользователя базы данных, что может приводить к ошибкам.

> **Подсказка**
>
> В журнале сервера может содержаться больше информации, чем в выдаваемых клиенту сообщениях об ошибке аутентификации, поэтому, если вас интересуют причины сбоя, проверьте журнал сервера.

Глава 20. Роли базы данных

Postgres Pro использует концепцию ролей (*roles*) для управления разрешениями на доступ к базе данных. Роль можно рассматривать как пользователя базы данных или как группу пользователей, в зависимости от того, как роль настроена. Роли могут владеть объектами базы данных (например, таблицами и функциями) и выдавать другим ролям разрешения на доступ к этим объектам, управляя тем, кто имеет доступ и к каким объектам. Кроме того, можно предоставить одной роли *членство* в другой роли, таким образом одна роль может использовать права других ролей.

Концепция ролей включает в себя концепцию пользователей («users») и групп («groups»). До версии 8.1 в PostgreSQL пользователи и группы были отдельными сущностями, но теперь есть только роли. Любая роль может использоваться в качестве пользователя, группы, и того и другого.

В этой главе описывается как создавать и управлять ролями. Дополнительную информацию о правах доступа ролей к различным объектам баз данных можно найти в Разделе 5.6.

20.1. Роли базы данных

Роли базы данных концептуально полностью отличаются от пользователей операционной системы. На практике поддержание соответствия между ними может быть удобным, но не является обязательным. Роли базы данных являются глобальными для всей установки кластера базы данных (не для отдельной базы данных). Для создания роли используется команда SQL CREATE ROLE:

```
CREATE ROLE имя;
```

Здесь *имя* соответствует правилам именования идентификаторов SQL: либо обычное, без специальных символов, либо в двойных кавычках. (На практике, к команде обычно добавляются другие указания, такие как `LOGIN`. Подробнее об этом ниже.) Для удаления роли используется команда DROP ROLE:

```
DROP ROLE имя;
```

Для удобства поставляются программы createuser и dropuser, которые являются обёртками для этих команд SQL и вызываются из командной строки оболочки ОС:

```
createuser имя
dropuser имя
```

Для получения списка существующих ролей, рассмотрите `pg_roles` системного каталога, например:

```
SELECT rolname FROM pg_roles;
```

Метакоманда `\du` программы psql также полезна для получения списка существующих ролей.

Для начальной настройки кластера базы данных, система сразу после инициализации всегда содержит одну предопределённую роль. Эта роль является суперпользователем («superuser») и по умолчанию (если не изменено при запуске `initdb`) имеет такое же имя, как и пользователь операционной системы, инициализирующий кластер баз данных. Обычно эта роль называется `postgres`. Для создания других ролей, вначале нужно подключиться с этой ролью.

Каждое подключение к серверу базы данных выполняется под именем конкретной роли, и эта роль определяет начальные права доступа для команд, выполняемых в этом соединении. Имя роли для конкретного подключения к базе данных указывается клиентской программой характерным для неё способом, таким образом инициируя запрос на подключение. Например, программа `psql` для указания роли использует аргумент командной строки `-U`. Многие приложения предполагают, что по умолчанию нужно использовать имя пользователя операционной системы (включая `createuser` и `psql`). Поэтому часто бывает удобным поддерживать соответствие между именами ролей и именами пользователей операционной системы.

Список доступных для подключения ролей, который могут использовать клиенты, определяется конфигурацией аутентификации, как описывалось в Главе 19. (Поэтому клиент не ограничен

только ролью, соответствующей имени пользователя операционной системы, так же как и имя для входа может не соответствовать реальному имени.) Так как с ролью связан набор прав, доступных для клиента, в многопользовательской среде распределять права следует с осторожностью.

20.2. Атрибуты ролей

Роль базы данных может иметь атрибуты, определяющие её полномочия и взаимодействие с системой аутентификации клиентов.

Право подключения

Только роли с атрибутом `LOGIN` могут использоваться для начального подключения к базе данных. Роль с атрибутом `LOGIN` можно рассматривать как пользователя базы данных. Для создания такой роли можно использовать любой из вариантов:

```
CREATE ROLE имя LOGIN;
CREATE USER имя;
```

(Команда `CREATE USER` эквивалентна `CREATE ROLE` за исключением того, что `CREATE USER` по умолчанию предполагает атрибут `LOGIN`, в то время как `CREATE ROLE` нет.)

Статус суперпользователя

Суперпользователь базы данных обходит все проверки прав доступа, за исключением права на вход в систему. Это опасная привилегия и она не должна использоваться небрежно. Лучше всего выполнять большую часть работы не как суперпользователь. Для создания нового суперпользователя используется `CREATE ROLE имя SUPERUSER`. Это нужно выполнить из под роли, которая также является суперпользователем.

Создание базы данных

Роль должна явно иметь разрешение на создание базы данных (за исключением суперпользователей, которые пропускают все проверки). Для создания такой роли используется `CREATE ROLE имя CREATEDB`.

Создание роли

Роль должна явно иметь разрешение на создание других ролей (за исключением суперпользователей, которые пропускают все проверки). Для создания такой роли используется `CREATE ROLE имя CREATEROLE`. Роль с правом `CREATEROLE` может не только создавать, но и изменять и удалять другие роли, а также выдавать и отзывать членство в ролях. Однако для создания, изменения, удаления ролей суперпользователей и изменения членства в них требуется иметь статус суперпользователя; права `CREATEROLE` в таких случаях недостаточно.

Запуск репликации

Роль должна иметь явное разрешение на запуск потоковой репликации (за исключением суперпользователей, которые пропускают все проверки). Роль, используемая для потоковой репликации, также должна иметь атрибут `LOGIN`. Для создания такой роли используется `CREATE ROLE имя REPLICATION LOGIN`.

Пароль

Пароль имеет значение, если метод аутентификации клиентов требует, чтобы пользователи предоставляли пароль при подключении к базе данных. Методы аутентификации `password` и `md5` используют пароли. База данных и операционная система используют раздельные пароли. Пароль указывается при создании роли: `CREATE ROLE имя PASSWORD 'строка'`.

Наследование прав

По умолчанию роль получает разрешение наследовать права той роли, членом которой она является. Однако можно создать роль и без такого разрешения, выполнив `CREATE ROLE имя NOINHERIT`.

Игнорирование защиты на уровне строк

Разрешение обходить все политики защиты на уровне строк (RLS) нужно давать роли явно. Создать роль с таким разрешением можно, выполнив `CREATE ROLE имя BYPASSRLS` от имени суперпользователя.

Ограничение соединений

Для роли можно ограничить число одновременных соединений. -1 (по умолчанию) означает отсутствие ограничения. Для создания роли с ограничением используется `CREATE ROLE name CONNECTION LIMIT 'число'`.

Атрибуты ролей могут быть изменены после создания командой `ALTER ROLE`. Более детальная информация в справке по командам CREATE ROLE и ALTER ROLE.

> **Подсказка**
>
> Рекомендуется создать роль с правами `CREATEDB` и `CREATEROLE`, но не суперпользователя, и в последующем использовать её для управления базами данных и ролями. Такой подход позволит избежать опасностей, связанных с использованием полномочий суперпользователя для задач, которые их не требуют.

На уровне ролей можно устанавливать многие конфигурационные параметры времени выполнения, описанные в Главе 18. Например, если по некоторым причинам всякий раз при подключении к базе данных требуется отключить использование индексов (подсказка: плохая идея) можно выполнить:

```
ALTER ROLE myname SET enable_indexscan TO off;
```

Установленное значение параметра будет сохранено (но не будет применено сразу). Для последующих подключений с этой ролью это будет выглядеть как выполнение команды `SET enable_indexscan TO off` перед началом сеанса. Но это только значение по умолчанию; в течение сеанса этот параметр можно изменить. Для удаления установок на уровне ролей для параметров конфигурации используется `ALTER ROLE имя_роли RESET имя_переменной`. Обратите внимание, что установка параметров конфигурации на уровне роли без права `LOGIN` лишено смысла, т. к. они никогда не будут применены.

20.3. Членство в роли

Часто бывает удобно сгруппировать пользователей для упрощения управления правами: права можно выдавать для всей группы и у всей группы забирать. В Postgres Pro для этого создаётся роль, представляющая группу, а затем *членство* в этой группе выдаётся ролям индивидуальных пользователей.

Для настройки групповой роли сначала нужно создать саму роль:

```
CREATE ROLE имя;
```

Обычно групповая роль не имеет атрибута `LOGIN`, хотя при желании его можно установить.

После того как групповая роль создана, в неё можно добавлять или удалять членов, используя команды GRANT и REVOKE:

```
GRANT групповая_роль TO роль1, ... ;
REVOKE групповая_роль FROM роль1, ... ;
```

Членом роли может быть и другая групповая роль (потому что в действительности нет никаких различий между групповыми и не групповыми ролями). При этом база данных не допускает замыкания членства по кругу. Также не допускается управление членством роли `PUBLIC` в других ролях.

Члены групповой роли могут использовать её права двумя способами. Во-первых, каждый член группы может явно выполнить SET ROLE, чтобы временно «стать» групповой ролью. В этом состоянии сеанс базы данных использует полномочия групповой роли вместо оригинальной роли, под которой был выполнен вход в систему. При этом для всех создаваемых объектов базы данных владельцем считается групповая, а не оригинальная роль. Во-вторых, роли, имеющие атрибут INHERIT, автоматически используют права всех ролей, членами которых они являются, в том числе и унаследованные этими ролями права. Например:

```
CREATE ROLE joe LOGIN INHERIT;
CREATE ROLE admin NOINHERIT;
CREATE ROLE wheel NOINHERIT;
GRANT admin TO joe;
GRANT wheel TO admin;
```

После подключения с ролью joe сеанс базы данных будет использовать права, выданные напрямую joe, и права, выданные роли admin, так как joe «наследует» права admin. Однако права, выданные wheel, не будут доступны, потому что, хотя joe неявно и является членом wheel, это членство получено через роль admin, которая имеет атрибут NOINHERIT. После выполнения команды:

```
SET ROLE admin;
```

сеанс будет использовать только права, назначенные admin, а права, назначенные роли joe, не будут доступны. После выполнения команды:

```
SET ROLE wheel;
```

сеанс будет использовать только права, выданные wheel, а права joe и admin не будут доступны. Начальный набор прав можно получить любой из команд:

```
SET ROLE joe;
SET ROLE NONE;
RESET ROLE;
```

Примечание

Команда SET ROLE в любой момент разрешает выбрать любую роль, прямым или косвенным членом которой является оригинальная роль, под которой был выполнен вход в систему. Поэтому в примере выше не обязательно сначала становиться admin перед тем как стать wheel.

Примечание

В стандарте SQL есть чёткое различие между пользователями и ролями. При этом пользователи, в отличие от ролей, не наследуют права автоматически. Такое поведение может быть получено в Postgres Pro, если для ролей, используемых как роли в стандарте SQL, устанавливать атрибут INHERIT, а для ролей-пользователей в стандарте SQL — атрибут NOINHERIT. Однако в Postgres Pro все роли по умолчанию имеют атрибут INHERIT. Это сделано для обратной совместимости с версиями до 8.1, в которых пользователи всегда могли использовать права групп, членами которых они являются.

Атрибуты роли LOGIN, SUPERUSER, CREATEDB и CREATEROLE можно рассматривать как особые права, но они никогда не наследуются как обычные права на объекты базы данных. Чтобы ими воспользоваться, необходимо переключиться на роль, имеющую этот атрибут, с помощью команды SET ROLE. Продолжая предыдущий пример, можно установить атрибуты CREATEDB и CREATEROLE для роли admin. Затем при входе с ролью joe, получить доступ к этим правам будет возможно только после выполнения SET ROLE admin.

Для удаления групповой роли используется DROP ROLE:

```
DROP ROLE имя;
```

Любое членство в групповой роли будет автоматически отозвано (в остальном на членов этой роли это никак не повлияет).

20.4. Удаление ролей

Так как роли могут владеть объектами баз данных и иметь права доступа к объектам других, удаление роли не сводится к немедленному действию DROP ROLE. Сначала должны быть удалены и переданы другим владельцами все объекты, принадлежащие роли; также должны быть отозваны все права, данные роли.

Владение объектами можно передавать в индивидуальном порядке, применяя команду ALTER, например:

```
ALTER TABLE bobs_table OWNER TO alice;
```

Кроме того, для переназначения какой-либо другой роли владения сразу всеми объектами, принадлежащими удаляемой роли, можно применить команду REASSIGN OWNED. Так как REASSIGN OWNED не может обращаться к объектам в других базах данных, её необходимо выполнить в каждой базе, которая содержит объекты, принадлежащие этой роли. (Заметьте, что первая такая команда REASSIGN OWNED изменит владельца для всех разделяемых между базами объектов, то есть для баз данных или табличных пространств, принадлежащих удаляемой роли.)

После того как все ценные объекты будут переданы новым владельцам, все оставшиеся объекты, принадлежащие удаляемой роли, могут быть удалены с помощью команды DROP OWNED. И эта команда не может обращаться к объектам в других базах данных, так что её нужно запускать в каждой базе, которая содержит объекты, принадлежащие роли. Также заметьте, что DROP OWNED не удаляет табличные пространства или базы данных целиком, так что это необходимо сделать вручную, если роли принадлежат базы или табличные пространства, не переданные новым владельцам.

DROP OWNED также удаляет все права, которые даны целевой роли для объектов, не принадлежащих ей. Так как REASSIGN OWNED такие объекты не затрагивает, обычно необходимо запустить и REASSIGN OWNED, и DROP OWNED (в этом порядке!), чтобы полностью ликвидировать зависимости удаляемой роли.

С учётом этого, общий рецепт удаления роли, которая владела объектами, вкратце таков:

```
REASSIGN OWNED BY doomed_role TO successor_role;
DROP OWNED BY doomed_role;
-- повторить предыдущие команды для каждой базы в кластере
DROP ROLE doomed_role;
```

Когда не все объекты нужно передать одному новому владельцу, лучше сначала вручную отработать исключения, а в завершение выполнить показанные выше действия.

При попытке выполнить DROP ROLE для роли, у которой сохраняются зависимые объекты, будут выданы сообщения, говорящие, какие объекты нужно передать другому владельцу или удалить.

20.5. Предопределённые роли

В Postgres Pro имеется набор предопределённых ролей, которые дают доступ к некоторым часто востребованным, но не общедоступным функциям и данным. Администраторы могут назначать (GRANT) эти роли пользователям и/или ролям в своей среде, таким образом открывая этим пользователям доступ к указанной функциональности и информации.

Имеющиеся предопределённые роли описаны в Таблице 20.1. Заметьте, что конкретные разрешения для каждой из предопределённых ролей в будущем могут изменяться по мере добавления дополнительной функциональности. Администраторы должны следить за этими изменениями, просматривая замечания к выпускам.

Таблица 20.1. Предопределённые роли

Роль	Разрешаемый доступ
pg_read_all_settings	Читать все конфигурационные переменные, даже те, что обычно видны только суперпользователям.
pg_read_all_stats	Читать все представления pg_stat_* и использовать различные расширения, связанные со статистикой, даже те, что обычно видны только суперпользователям.
pg_stat_scan_tables	Выполнять функции мониторинга, которые могут устанавливать блокировки ACCESS SHARE в таблицах, возможно, на длительное время.
pg_monitor	Читать/выполнять различные представления и функции для мониторинга. Эта роль включена в роли pg_read_all_settings, pg_read_all_stats и pg_stat_scan_tables.
pg_signal_backend	Передавать другим обслуживающим процессам сигналы для отмены запроса или завершения сеанса.

Роли pg_monitor, pg_read_all_settings, pg_read_all_stats и pg_stat_scan_tables созданы для того, чтобы администраторы могли легко настроить роль для мониторинга сервера БД. Эти роли наделяют своих членов набором общих прав, позволяющих читать различные полезные параметры конфигурации, статистику и другую системную информацию, что обычно доступно только суперпользователям.

Роль pg_signal_backend предназначена для того, чтобы администраторы могли давать доверенным, но не имеющим прав суперпользователя ролям право посылать сигналы другим обслуживающим процессам. В настоящее время эта роль позволяет передавать сигналы для отмены запроса, выполняющегося в другом процессе, или для завершения сеанса. Однако пользователь, включённый в эту роль, не может отправлять сигналы процессам, принадлежащим суперпользователям. См. Подраздел 9.26.2.

Распределяя эти роли, следует проявлять осторожность, чтобы они использовались только для целей мониторинга.

Администраторы могут давать пользователям доступ к этим ролям, используя команду GRANT. Например:

```
GRANT pg_signal_backend TO admin_user;
```

20.6. Безопасность функций

Функции, триггеры и политики защиты на уровне строк позволяют пользователям внедрять код в обслуживающие процессы, который может быть непреднамеренно выполнен другими пользователями. Таким образом эти механизмы позволяют пользователям запускать «троянский код» относительно просто. Лучшая защита от этого — строгое ограничение круга лиц, которые могут создавать объекты. Там где это невозможно, пишите запросы так, чтобы они ссылались только на объекты с доверенными владельцами. Удалите из search_path схему public и любые другие схемы, в которых могут создавать объекты недоверенные пользователи.

Функции выполняются внутри серверного процесса с полномочиями пользователя операционной системы, запускающего сервер базы данных. Если используемый для функций язык программирования разрешает неконтролируемый доступ к памяти, то это даёт возможность изменить внутренние структуры данных сервера. Таким образом, помимо всего прочего, такие функции могут обойти ограничения доступа к системе. Языки программирования, допускающие

такой доступ, считаются «недоверенными» и создавать функции на этих языках Postgres Pro разрешает только суперпользователям.

Глава 21. Управление базами данных

Каждый работающий экземпляр сервера Postgres Pro обслуживает одну или несколько баз данных. Поэтому базы данных представляют собой вершину иерархии SQL-объектов («объектов базы данных»). Данная глава описывает свойства баз данных, процессы создания, управления и удаления.

21.1. Обзор

Некоторые объекты, включая роли, базы данных и табличные пространства, определяются на уровне кластера и сохраняются в табличном пространстве `pg_global`. Внутри кластера существуют базы данных, которые отделены друг от друга, но могут обращаться к объектам уровня кластера. Внутри каждой базы данных имеются схемы, содержащие такие объекты, как таблицы и функции. Таким образом, полная иерархия выглядит следующим образом: кластер, база данных, схема, таблица (или иной объект, например функция).

При подключении к серверу баз данных клиент должен указать имя базы в запросе подключения. Обращаться к нескольким базам через одно подключение нельзя, однако клиенты могут открыть несколько подключений к одной базе или к разным. Безопасность на уровне базы обеспечивают две составляющие: управление подключениями (см. Раздел 19.1), которое осуществляется на уровне соединения, и управление доступом к объектам (см. Раздел 5.6), для которого реализована система прав. Обёртки сторонних данных (см. postgres_fdw) позволяют создать в одной базе данных объекты, скрывающие за собой объекты в других базах или кластерах. Подобную же функциональность предоставляет и более старый модуль dblink (см. dblink). По умолчанию все пользователи могут подключаться ко всем базам данных, используя все методы подключения.

В случаях, когда в кластере Postgres Pro планируется размещать данные несвязанных проектов или с ним будут работать пользователи, которые в принципе не должны никак взаимодействовать, рекомендуется использовать отдельные базы данных и организовать управление подключением и доступом к объектам соответствующим образом. Если же проекты или пользователи взаимосвязаны и должны иметь возможность использовать ресурсы друг друга, они должны размещаться в одной базе данных, но, возможно, в отдельных схемах. Таким образом будет создана модульная структура с изолированными пространствами имён и управлением доступа. Подробнее об управлении схемами рассказывается в Разделе 5.8.

Хотя в одном кластере можно создать несколько баз данных, прежде чем это делать, рекомендуется тщательно взвесить все связанные с этим риски и ограничения. В частности, наличие общего WAL (см. Главу 29) может повлиять на возможности резервного копирования и восстановления данных. Отдельные базы в кластере изолированы друг от друга с точки зрения пользователя, но с точки зрения администратора баз данных они тесно связаны.

Базы данных создаются командой `CREATE DATABASE` (см. Раздел 21.2), а удаляются командой `DROP DATABASE` (см. Раздел 21.5). Список существующих баз данных можно посмотреть в системном каталоге `pg_database`, например,

```
SELECT datname FROM pg_database;
```

Метакоманда `\l` или ключ `-l` командной строки приложения psql также позволяют вывести список существующих баз данных.

> **Примечание**
>
> Стандарт SQL называет базы данных «каталогами», но на практике у них нет отличий.

21.2. Создание базы данных

Для создания базы данных сервер Postgres Pro должен быть развёрнут и запущен (см. Раздел 17.3).

База данных создаётся SQL-командой CREATE DATABASE:

```
CREATE DATABASE имя;
```

где *имя* подчиняется правилам именования идентификаторов SQL. Текущий пользователь автоматически назначается владельцем. Владелец может удалить свою базу, что также приведёт к удалению всех её объектов, в том числе, имеющих других владельцев.

Создание баз данных это привилегированная операция. Как предоставить права доступа, описано в Разделе 20.2.

Поскольку для выполнения команды `CREATE DATABASE` необходимо подключение к серверу базы данных, возникает вопрос как создать самую первую базу данных. Первая база данных всегда создаётся командой `initdb` при инициализации пространства хранения данных (см. Раздел 17.2.) Эта база данных называется `postgres`. Далее для создания первой «обычной» базы данных можно подключиться к `postgres`.

Вторая база данных `template1`, также создаётся во время инициализации кластера. При каждом создании новой базы данных в рамках кластера по факту производится клонирование шаблона `template1`. При этом любые изменения, сделанные в `template1` распространяются на все созданные впоследствии базы данных. Следует избегать создания объектов в `template1`, за исключением ситуации, когда их необходимо автоматически добавлять в новые базы. Более подробно в Разделе 21.3.

Для удобства, есть утилита командной строки для создания баз данных, `createdb`.

```
createdb dbname
```

Утилита `createdb` не делает ничего волшебного, она просто подключается к базе данных `postgres` и выполняет ранее описанную SQL-команду `CREATE DATABASE`. Подробнее о её вызове можно узнать в createdb. Обратите внимание, что команда `createdb` без параметров создаст базу данных с именем текущего пользователя.

> **Примечание**
>
> Глава 19 содержит информацию о том, как ограничить права на подключение к заданной базе данных.

Иногда необходимо создать базу данных для другого пользователя и назначить его владельцем, чтобы он мог конфигурировать и управлять ею. Для этого используйте одну из следующих команд:

```
CREATE DATABASE имя_базы OWNER имя_роли;
```

из среды SQL, или:

```
createdb -O имя_роли имя_базы
```

из командной строки ОС. Лишь суперпользователь может создавать базы данных для других (для ролей, членом которых он не является).

21.3. Шаблоны баз данных

По факту команда `CREATE DATABASE` выполняет копирование существующей базы данных. По умолчанию копируется стандартная системная база `template1`. Таким образом, `template1` это шаблон, на основе которого создаются новые базы. Если добавить объекты в `template1`, то впоследствии они будут копироваться в новые базы данных. Это позволяет внести изменения в стандартный набор объектов. Например, если в `template1` установить процедурный язык PL/Perl, то он будет доступен в новых базах без дополнительных действий.

Также существует вторая системная база `template0`. При инициализации она содержит те же самые объекты, что и `template1`, предопределённые в рамках устанавливаемой версии Postgres

Pro. Не нужно вносить никаких изменений в template0 после инициализации кластера. Если в команде CREATE DATABASE указать на необходимость копирования template0 вместо template1, то на выходе можно получить «чистую» пользовательскую базу данных без изменений, внесённых в template1. Это удобно, когда производится восстановление из дампа данных с помощью утилиты pg_dump: скрипт дампа лучше выполнять в чистую базу, во избежание каких-либо конфликтов с объектами, которые могли быть добавлены в template1.

Другая причина, для копирования template0 вместо template1 заключается в том, что можно указать новые параметры локали и кодировку при копировании template0, в то время как для копий template1 они не должны меняться. Это связано с тем, что template1 может содержать данные в специфических кодировках и локалях, в отличие от template0.

Для создания базы данных на основе template0, используйте:

```
CREATE DATABASE dbname TEMPLATE template0;
```

из среды SQL, или:

```
createdb -T template0 dbname
```

из командной строки ОС.

Можно создавать дополнительные шаблоны баз данных, и, более того, можно копировать любую базу данных кластера, если указать её имя в качестве шаблона в команде CREATE DATABASE. Важно понимать, что это (пока) не рассматривается в качестве основного инструмента для реализации возможности «COPY DATABASE». Важным является то, что при копировании все сессии к копируемой базе данных должны быть закрыты. CREATE DATABASE выдаст ошибку, если есть другие подключения; во время операции копирования новые подключения к этой базе данных не разрешены.

В таблице pg_databaseесть два полезных флага для каждой базы данных: столбцы datistemplate и datallowconn. datistemplate указывает на факт того, что база данных может выступать в качестве шаблона в команде CREATE DATABASE. Если флаг установлен, то для пользователей с правом CREATEDB клонирование доступно; если флаг не установлен, то лишь суперпользователь и владелец базы данных могут её клонировать. Если datallowconn не установлен, то новые подключения к этой базе не допустимы (однако текущие сессии не закрываются при сбросе этого флага). База template0 обычно помечена как datallowconn = false для избежания любых её модификаций. И template0, и template1 всегда должны быть помечены флагом datistemplate = true.

> **Примечание**
>
> template1 и template0 не выделены как-то особенно, кроме того факта, что template1 используется по умолчанию в команде CREATE DATABASE. Например, можно удалить template1 и безболезненно создать заново из template0. Это можно посоветовать в случае, если template1 был замусорен. (Чтобы удалить template1, необходимо сбросить флаг pg_database.datistemplate = false.)
>
> База данных postgres также создаётся при инициализации кластера. Она используется пользователями и приложениями для подключения по умолчанию. Представляет собой всего лишь копию template1, и может быть удалена и повторно создана при необходимости.

21.4. Конфигурирование баз данных

Обратившись к Главе 18 можно выяснить, что сервер Postgres Pro имеет множество параметров конфигурации времени выполнения. Можно выставить специфичные для базы данных значения по умолчанию.

Например, если по какой-то причине необходимо выключить GEQO оптимизатор в какой-то из баз, то можно, либо выключить его для всех баз данных одновременно, либо убедиться, что все

клиенты заботятся об этом, выполняя команду `SET geqo TO off`. Для того чтобы это действовало по умолчанию в конкретной базе данных, необходимо выполнить команду:

```
ALTER DATABASE mydb SET geqo TO off;
```

Установка сохраняется, но не применяется тотчас. В последующих подключениях к этой базе данных, эффект будет таким, будто перед началом сессии была выполнена команда `SET geqo TO off;`. Стоит обратить внимание, что пользователь по-прежнему может изменять этот параметр во время сессии; ведь это просто значение по умолчанию. Чтобы сбросить такое установленное значение, используйте `ALTER DATABASE` *dbname* `RESET` *varname*.

21.5. Удаление базы данных

Базы данных удаляются командой DROP DATABASE:

```
DROP DATABASE имя;
```

Лишь владелец базы данных или суперпользователь могут удалить базу. При удалении также удаляются все её объекты. Удаление базы данных это необратимая операция.

Невозможно выполнить команду `DROP DATABASE` пока существует хоть одно подключение к заданной базе. Однако можно подключиться к любой другой, в том числе и `template1`. `template1` может быть единственной возможностью при удалении последней пользовательской базы данных кластера.

Также существует утилита командной строки для удаления баз данных dropdb:

```
dropdb dbname
```

(В отличие от команды `createdb` утилита не использует имя текущего пользователя по умолчанию).

21.6. Табличные пространства

Табличные пространства в Postgres Pro позволяют администраторам организовать логику размещения файлов объектов базы данных в файловой системе. К однажды созданному табличному пространству можно обращаться по имени на этапе создания объектов.

Табличные пространства позволяют администратору управлять дисковым пространством для инсталляции Postgres Pro. Это полезно минимум по двум причинам. Во-первых, это нехватка места в разделе, на котором был инициализирован кластер и невозможность его расширения. Табличное пространство можно создать в другом разделе и использовать его до тех пор, пока не появится возможность переконфигурирования системы.

Во-вторых, табличные пространства позволяют администраторам оптимизировать производительность согласно бизнес-процессам, связанным с объектами базы данных. Например, часто используемый индекс можно разместить на очень быстром и надёжном, но дорогом SSD-диске. В то же время таблица с архивными данными, которые редко используются и скорость к доступа к ним не важна, может быть размещена в более дешёвом и медленном хранилище.

> **Предупреждение**
>
> Несмотря на внешнее размещение относительно основного каталога хранения данных Postgres Pro, табличные пространства являются неотъемлемой частью кластера и *не могут* трактоваться, как самостоятельная коллекция файлов данных. Они зависят от метаданных, расположенных в главном каталоге, и потому не могут быть подключены к другому кластеру, или копироваться по отдельности. Также, в случае потери табличного пространства (при удалении файлов, сбое диска и т. п.), кластер может оказаться недоступным или не сможет запуститься. Таким образом, при размещении табличного пространства во временной файловой системе, например, в RAM-диске, возникает угроза надёжности всего кластера.

Для создания табличного пространства используется команда CREATE TABLESPACE, например::

```
CREATE TABLESPACE fastspace LOCATION '/ssd1/postgresql/data';
```

Каталог должен существовать, быть пустым и принадлежать пользователю ОС, под которым запущен Postgres Pro. Все созданные впоследствии объекты, принадлежащие целевому табличному пространству, будут храниться в файлах расположенных в этом каталоге. Каталог не должен размещаться на съёмных или устройствах временного хранения, так как кластер может перестать функционировать из-за потери этого пространства.

> **Примечание**
>
> Обычно нет смысла создавать более одного пространства на одну логическую файловую систему, так как нет возможности контролировать расположение отдельных файлов в файловой системе. Однако Postgres Pro не накладывает никаких ограничений в этом отношении, и более того, напрямую не заботится о точках монтирования файловой системы. Просто осуществляется хранение файлов в указанных каталогах.

Создавать табличное пространство должен суперпользователь базы данных, но после этого можно разрешить обычным пользователям его использовать. Для этого необходимо предоставить привилегию `CREATE` на табличное пространство.

Таблицы, индексы и целые базы данных могут храниться в отдельных табличных пространствах. Для этого пользователь с правом `CREATE` на табличное пространство должен указать его имя в качестве параметра соответствующей команды. Например, далее создаётся таблица в табличном пространстве `space1`:

```
CREATE TABLE foo(i int) TABLESPACE space1;
```

Как вариант, используйте параметр default_tablespace:

```
SET default_tablespace = space1;
CREATE TABLE foo(i int);
```

Когда `default_tablespace` имеет значение отличное от пустой строки, он будет использоваться неявно в качестве значения параметра `TABLESPACE` в командах `CREATE TABLE` и `CREATE INDEX`, если в самой команде не задано иное.

Существует параметр temp_tablespaces, который указывает на размещение временных таблиц и индексов, а также файлов, создаваемых, например, при операциях сортировки больших наборов данных. Предпочтительнее, в качестве значения этого параметра, указывать не одно имя, а список из нескольких табличных пространств. Это поможет распределить нагрузку, связанную с временными объектами, по различным табличным пространствам. При каждом создании временного объекта будет случайным образом выбираться имя из указанного списка табличных пространств.

Табличное пространство, связанное с базой данных, также используется для хранения её системных каталогов. Более того, это табличное пространство используется по умолчанию для таблиц, индексов и временных файлов, создаваемых в базе данных, если не указано иное в выражении `TABLESPACE`, или переменной `default_tablespace`, или `temp_tablespaces` (соответственно). Если база данных создана без указания конкретного табличного пространства, то используется пространство, к которому принадлежит копируемый шаблон.

При инициализации кластера автоматически создаются два табличных пространства. Табличное пространство `pg_global` используется для общих системных каталогов. Табличное пространство `pg_default` используется по умолчанию для баз данных `template1` и `template0` (в свою очередь, также является пространством по умолчанию для других баз данных, пока не будет явно указано иное в выражении `TABLESPACE` команды `CREATE DATABASE`).

После создания, табличное пространство можно использовать в рамках любой базы данных, при условии, что у пользователя имеются необходимые права. Это означает, что табличное

пространство невозможно удалить до тех пор, пока не будут удалены все объекты баз данных, использующих это пространство.

Для удаления пустого табличного пространства используйте команду DROP TABLESPACE.

Чтобы получить список табличных пространств можно сделать запрос к системному каталогу pg_tablespace, например,

```
SELECT spcname FROM pg_tablespace;
```

Метакоманда \db утилиты psql также позволяет отобразить список существующих табличных пространств.

Postgres Pro использует символические ссылки для упрощения реализации табличных пространств. Это означает, что табличные пространства могут использоваться *только* в системах, поддерживающих символические ссылки.

Каталог $PGDATA/pg_tblspc содержит символические ссылки, которые указывают на внешние табличные пространства кластера. Хоть и не рекомендуется, но возможно регулировать табличные пространства вручную, переопределяя эти ссылки. Ни при каких обстоятельствах эти операции нельзя проводить, пока запущен сервер баз данных. Обратите внимание, что в версии PostgreSQL 9.1 и более ранних также необходимо обновить информацию в pg_tablespace о новых расположениях. (Если это не сделать, то pg_dump будет продолжать выводить старые расположения табличных пространств.)

Глава 22. Локализация

Данная глава описывает доступные возможности локализации с точки зрения администратора. Postgres Pro поддерживает два средства локализации:

- Использование средств локализации операционной системы для обеспечения определяемых локалью порядка правил сортировки, форматирования чисел, перевода сообщений и прочих аспектов. Это рассматривается в Разделе 22.1 и Разделе 22.2.
- Обеспечение возможностей использования различных кодировок для хранения текста на разных языках и перевода кодировок между клиентом и сервером. Это рассматривается в Разделе 22.3.

22.1. Поддержка языковых стандартов

Поддержка *языковых стандартов* в приложениях относится к культурным предпочтениям, которые касаются алфавита, порядка сортировки, форматирования чисел и т. п. Postgres Pro использует соответствующие стандартам ISO C и POSIX возможности локали, предоставляемые операционной системой сервера. За дополнительной информацией обращайтесь к документации вашей системы.

22.1.1. Обзор

Поддержка локали автоматически инициализируется, когда кластер базы данных создаётся при помощи `initdb`. `initdb` инициализирует кластер баз данных по умолчанию с локалью из окружения выполнения. Поэтому, если ваша система уже использует локаль, которую вы хотите выбрать для вашего кластера баз данных, вам не требуется дополнительно совершать никаких действий. Если вы хотите использовать другую локаль (или вы точно не знаете, какая локаль используется в вашей системе), вы можете указать для `initdb`, какую именно локаль использовать, задав параметр `--locale`. Например:

```
initdb --locale=ru_RU
```

Данный пример для Unix-систем задаёт русский язык (`ru`), на котором говорят в России (RU) Другими вариантами могут быть `en_US` (американский английский) и `fr_CA` (канадский французский). Если в языковом окружении может использоваться более одного набора символов, значение может принимать вид *`language_territory.codeset`*. Например, `fr_BE.UTF-8` обозначает французский язык (fr), на котором говорят в Бельгии (BE), с кодировкой UTF-8.

То, какие локали и под какими именами доступны на вашей системе, зависит от того, что было включено в операционную систему производителем и что из этого было установлено. В большинстве Unix-систем команда `locale -a` выведет список доступных локалей. Windows использует более развёрнутые имена локалей, такие как `German_Germany` или `Russian_Russia.1251`, но принципы остаются теми же.

Иногда целесообразно объединить правила из различных локалей, например, использовать английские правила сравнения и испанские сообщения. Для этой цели существует набор категорий локали, каждая из которых управляет только определёнными аспектами правил локализации:

`LC_COLLATE`	Порядок сортировки строк
`LC_CTYPE`	Классификация символов (Что представляет собой буква? Каков её эквивалент в верхнем регистре?)
`LC_MESSAGES`	Язык сообщений
`LC_MONETARY`	Форматирование валютных сумм
`LC_NUMERIC`	Форматирование чисел

LC_TIME	Форматирование даты и времени

Эти имена категорий initdb принимает в качестве имён соответствующих параметров, позволяющих переопределить выбор локали в определённой категории. Например, чтобы настроить локаль на канадский французский, но при этом использовать американские правила форматирования денежных сумм, используйте initdb --locale=fr_CA --lc-monetary=en_US.

Если вы хотите, чтобы система работала без языковой поддержки, используйте специальное имя локали C либо эквивалентное ему POSIX.

Значения некоторых категорий локали должны быть заданы при создании базы данных. Вы можете использовать различные параметры локали для различных баз данных, но после создания базы вы уже не сможете изменить их для этой базы данных. LC_COLLATE и LC_CTYPE являются этими категориями. Они влияют на порядок сортировки в индексах, поэтому они должны быть зафиксированы, иначе индексы на текстовых столбцах могут повредиться. (Однако можно смягчить эти ограничения через задание правил сравнения, как это описано в разделе Раздел 22.2.) Значения по умолчанию для этих категорий определяются при запуске initdb, и эти значения используются при создании новых баз данных, если другие значения не указаны явно в команде CREATE DATABASE.

Прочие категории локали вы можете изменить в любое время, настроив параметры конфигурации сервера, которые имеют такое же имя как и категории локали (подробнее см. Подраздел 18.11.2). Значения, выбранные через initdb, фактически записываются лишь в файл конфигурации postgresql.conf, чтобы использоваться по умолчанию при запуске сервера. Если вы удалите эти значения из postgresql.conf, сервер получит соответствующие значения из своей среды выполнения.

Обратите внимание на то, что поведение локали сервера определяется переменными среды, установленными на стороне сервера, а не средой клиента. Таким образом, необходимо правильно сконфигурировать локаль перед запуском сервера. Если же клиент и сервер работают с разными локалями, то сообщения, возможно, будут появляться на разных языках в зависимости от того, где они возникают.

> **Примечание**
>
> Когда мы говорим о наследовании локали от среды выполнения, это означает следующее для большинства операционных систем: для определённой категории локали, к примеру, для правил сортировки, следующие переменные среды анализируются в приведённом ниже порядке до тех пор, пока одна из них не окажется заданной: LC_ALL, LC_COLLATE (или переменная, относящаяся к соответствующей категории), LANG. Если ни одна их этих переменных среды не задана, значение локали устанавливается по умолчанию в C.
>
> Некоторые библиотеки локализации сообщений также обращаются к переменной среды LANGUAGE, которая заменяет все прочие параметры локализации при выборе языка сообщений. В случае затруднений, пожалуйста, воспользуйтесь документацией по вашей операционной системе, в частности, справкой по gettext.

Для того чтобы стал возможен перевод сообщений на язык, выбранный пользователем, NLS должен быть выбран на момент сборки (configure --enable-nls). В остальном поддержка локализации осуществляется автоматически.

22.1.2. Поведение

Локаль влияет на следующий функционал SQL:

- Порядок сортировки в запросах с использованием ORDER BY или стандартных операторах сравнения текстовых данных
- Функции upper, lower, и initcap

- Операторы поиска по шаблону (LIKE, SIMILAR TO, и регулярные выражения в стиле POSIX); локаль влияет как на поиск без учёта регистра, так и на классификацию знаков по классам символов регулярных выражений
- Семейство функций to_char
- Возможность использовать индексы с предложениями LIKE

Недостатком использования отличающихся от C или POSIX локалей в Postgres Pro является влияние на производительность. Это замедляет обработку символов и мешает LIKE использовать обычные индексы. По этой причине используйте локали только в том случае, если они действительно вам нужны.

В качестве обходного решения, которое позволит Postgres Pro пользоваться индексами с предложениями LIKE с использованием локали, отличной от C, существует несколько классов пользовательских операторов. Они позволяют создать индекс, который выполняет строгое посимвольное сравнение, игнорируя правила сравнения, соответствующие локали. За дополнительными сведениями обратитесь к Разделу 11.9. Ещё один подход заключается в создании индексов с помощью правил сортировки C, как было сказано в Разделе 22.2.

22.1.3. Проблемы

Если поддержка локализации не работает в соответствии с объяснением, данным выше, проверьте, насколько корректна конфигурация поддержки локализации в вашей операционной системе. Чтобы проверить, какие локали установлены на вашей системе, вы можете использовать команду locale -a, если ваша операционная система поддерживает это.

Проверьте, действительно ли Postgres Pro использует локаль, которую вы подразумеваете. Параметры LC_COLLATE и LC_CTYPE определяются при создании базы данных, и не могут быть изменены, за исключением случаев, когда создаётся новая база данных. Прочие параметры локали, включая LC_MESSAGES и LC_MONETARY первоначально определены средой, в которой запускается сервер, но могут быть оперативно изменены. Вы можете проверить текущие параметры локали с помощью команды SHOW.

Клиентские приложения, которые обрабатывают ошибки сервера, разбирая текст сообщения об ошибке, очевидно, столкнутся с проблемами, когда сообщения сервера будут на другом языке. Авторам таких приложений рекомендуется пользоваться системой кодов ошибок в качестве альтернативы.

22.2. Поддержка правил сортировки

Правила сортировки позволяют устанавливать порядок сортировки и особенности классификации символов в отдельных столбцах или даже при выполнении отдельных операций. Это смягчает последствия того, что параметры базы данных LC_COLLATE и LC_CTYPE невозможно изменить после её создания.

22.2.1. Основные понятия

Концептуально, каждое выражение с типом данных, к которому применяется сортировка, имеет правила сортировки. (Встроенными сортируемыми типами данных являются text, varchar, и char. Типы, определяемые в базе пользователем, могут также быть отмечены как сортируемые, и, конечно, домен на основе сортируемого типа данных является сортируемым.) Если выражение содержит ссылку на столбец, правила сортировки выражения определяются правилами сортировки столбца. Если выражение — константа, правилами сортировки являются стандартные правила для типа данных константы. Правила сортировки более сложных выражений являются производной от правил сортировки входящих в него частей, как описано ниже.

Правилами сортировки выражения могут быть правила сортировки «по умолчанию», что означает использование параметров локали, установленных для базы данных. Также возможно, что правила сортировки выражения могут не определиться. В таких случаях операции упорядочивания и другие операции, для которых необходимы правила сортировки, завершатся с ошибкой.

Когда база данных должна выполнить упорядочивание или классификацию символов, она использует правила сортировки выполняемого выражения. Это происходит, к примеру, с предложениями `ORDER BY` и такими вызовами функций или операторов как `<`. Правила сортировки, которые применяются в предложении `ORDER BY`, это просто правила ключа сортировки. Правила сортировки, применяемые к вызову функции или оператора, определяются их параметрами, как описано ниже. В дополнение к операциям сравнения, правила сортировки учитываются функциями, преобразующими регистр символов, такими как `lower`, `upper`, и `initcap`; операторами поиска по шаблону; и функцией `to_char` и связанными с ней.

При вызове функции или оператора правило сортировки определяется в зависимости от того, какие правила заданы для аргументов во время выполнения данной операции. Если результатом вызова функции или оператора является сортируемый тип данных, правила сортировки также используются во время разбора как определяемые правила сортировки функции или выражения оператора, когда для внешнего выражения требуется знание правил сортировки.

Определение правил сортировки выражения может быть неявным или явным. Это отличие влияет на то, как комбинируются правила сортировки, когда несколько разных правил появляются в выражении. Явное определение правил сортировки возникает, когда используется предложение `COLLATE`; все прочие варианты являются неявными. Когда необходимо объединить несколько правил сортировки, например, в вызове функции, используются следующие правила:

1. Если для одного из выражений-аргументов правило сортировки определено явно, то и для других аргументов явно задаваемое правило должно быть тем же, иначе возникнет ошибка. В случае присутствия явного определения правила сортировки, оно становится результирующим для всей операции.
2. В противном случае все входные выражения должны иметь одни и те же неявно определяемые правила сортировки или правила сортировки по умолчанию. Если присутствуют какие- либо правила сортировки, отличные от заданных по умолчанию, получаем результат комбинации правил сортировки. Иначе результатом станут правила сортировки, заданные по умолчанию.
3. Если среди входных выражений есть конфликтующие неявные правила сортировки, отличные от заданных по умолчанию, тогда комбинация рассматривается как имеющая неопределённые правила сортировки. Это не является условием возникновения ошибки, если вызываемой конкретной функции не требуются данные о правилах сортировки, которые ей следует применить. Если же такие данные требуются, это приведёт к ошибке во время выполнения.

В качестве примера рассмотрим данное определение таблицы:

```
CREATE TABLE test1 (
    a text COLLATE "ru_RU",
    b text COLLATE "es_ES",
    ...
);
```

Затем в

```
SELECT a < 'foo' FROM test1;
```

выполняется оператор сравнения `<` согласно правилам `ru_RU`, так как выражение объединяет неявно определяемые правила сортировки с правилами, заданными по умолчанию. Но в

```
SELECT a < ('foo' COLLATE "fr_FR") FROM test1;
```

сравнение выполняется с помощью правил `fr_FR`, так как явное определение правил сортировки переопределяет неявное. Кроме того, в запросе

```
SELECT a < b FROM test1;
```

анализатор запросов не может определить, какое правило сортировки использовать, поскольку столбцы `a` и `b` имеют конфликтующие неявные правила сортировки. Так как оператору `<` требуется знать, какое правило использовать, это приведёт к ошибке. Ошибку можно устранить, применив явное указание правил сортировки к любому из двух входных выражений. Например:

```
SELECT a < b COLLATE "ru_RU" FROM test1;
```

либо эквивалентное ему

```
SELECT a COLLATE "ru_RU" < b FROM test1;
```

С другой стороны, следующее выражение схожей структуры

```
SELECT a || b FROM test1;
```

не приводит к ошибке, поскольку для оператора `||` правила сортировки не имеют значения, так как результат не зависит от сортировки.

Правила сортировки, назначенные функции или комбинации входных выражений оператора, также могут быть применены к функции или результату оператора, если функция или оператор возвращают результат сортируемого типа данных. Так, в

```
SELECT * FROM test1 ORDER BY a || 'foo';
```

упорядочение будет происходить согласно правилам `ru_RU`. Но данный запрос:

```
SELECT * FROM test1 ORDER BY a || b;
```

приводит к ошибке, потому что, даже если оператору `||` не нужно знать правила сортировки, предложению `ORDER BY` это требуется. Как было сказано выше, конфликт может быть разрешён при помощи явного указания правил сортировки:

```
SELECT * FROM test1 ORDER BY a || b COLLATE "fr_FR";
```

22.2.2. Управление правилами сортировки

Правила сортировки представляют собой объект схемы SQL, который сопоставляет SQL-имя с локалью, реализуемой библиотекой, установленной в операционной системе. В определении правила сортировки задаётся *провайдер*, то есть библиотека, реализующая правило сортировки. Стандартный провайдер с именем `libc` использует системную библиотеку C и предоставляет её локали. Именно эти локали используются большинством утилит операционной системы. Также есть провайдер `icu`, который использует внешнюю библиотеку ICU. Локали ICU можно использовать, только если поддержка ICU была включена в конфигурации сборки Postgres Pro.

Правило сортировки, предоставляемое провайдером `libc`, сопоставляется с комбинацией параметров `LC_COLLATE` и `LC_CTYPE`, которую может принять системный вызов `setlocale()`. (Основная цель правила сортировки — настроить параметр `LC_COLLATE`, который управляет упорядочиванием символов. Однако на практике редко требуется иметь значение `LC_CTYPE`, отличное от `LC_COLLATE`, поэтому удобнее объединить их в одну сущность, и не создавать отдельную инфраструктуру для указания `LC_CTYPE` в выражениях.) Правила сортировки `libc` также связаны с кодировкой набора символов (см. Раздел 22.3). Одно и то же имя правила сортировки может существовать для разных кодировок.

Объект правила сортировки, предоставляемой провайдером `icu`, сопоставляется с именованным сортировщиком, реализуемым библиотекой ICU. ICU не поддерживает различные характеристики «collate» и «ctype», так что они всегда совпадают. Кроме того, правила сортировки ICU не зависят от кодировки, так что в базе данных будет всего одно правило сортировки ICU с определённым именем.

Вы можете задать провайдер основного правила сортировки в аргументах `--locale` и `--lc-collate` команд initdb или createdb следующим образом:

```
--locale=локаль[@провайдер]
--lc-collate=локаль[@провайдер]
```

Здесь параметр *провайдер* может принимать значение `icu` или `libc`, а *локаль* задаётся в формате `libc`. После символа `@` может быть задан только один провайдер. Значение `--lc-collate` переопределяет значение `--locale` независимо от того, задаётся ли в этих значениях провайдер.

Если вы не укажете провайдер, для локалей `C` и `POSIX` будет выбран провайдер `libc`, а для других провайдер выбирается так:

- `icu` на уровне кластера

- Провайдер основного правила сортировки из шаблона БД на уровне баз данных

> **Важно**
>
> Вы можете использовать провайдер `icu` только для локалей, которые поддерживаются библиотекой `libc` в вашей операционной системе и удовлетворяют всем ограничениям, присущим `icu`.

Когда вы запускаете сервер БД или подключаетесь к базе данных, Postgres Pro проверяет, поддерживается ли выбранный провайдер правил сортировки и версия правила сортировки по умолчанию. Узнать установленное для базы данных правило сортировки по умолчанию и провайдер правил сортировки можно в `pg_database.datcollate`. Для правил сортировки ICU также хранится их версия:

локаль`@`*провайдер*`[.`*версия*`]`

Перед запуском сервера Postgres Pro проверяет, был ли кластер инициализирован сервером с той же версией ICU, даже когда провайдер основного правила сортировки — не `icu`. Если версии библиотек ICU различаются, сервер Postgres Pro выдаёт соответствующее сообщение.

Перед подключением к базе данных, в которой основное правило сортировки использует `icu`, Postgres Pro сравнивает версию этого правила сортировки с версией, предоставляемой библиотекой ICU. Если версия меняется, и вы полагаете, что это изменение может повлиять на порядок сортировки ваших данных, перестройте объекты, использующие правило сортировки по умолчанию. Postgres Pro выдаёт предупреждения в случае несовпадения версий или если сравнить их невозможно по причине отсутствия в кластере информации о версии основного правила сортировки. Чтобы убрать эти предупреждения, вы можете выполнить команду `ALTER COLLATION "default" REFRESH VERSION`, как описано в ALTER COLLATION.

22.2.2.1. Стандартные правила сортировки

На всех платформах доступны правила сортировки под названием `default`, `C`, и `POSIX`. Дополнительные правила сортировки могут быть доступны в зависимости от поддержки операционной системы. Правило сортировки `default` использует значения `LC_COLLATE` и `LC_CTYPE`, заданные при создании базы данных. Правила сортировки `C` и `POSIX` определяют поведение, характерное для «традиционного C», в котором только знаки кодировки ASCII от «`A`» до «`Z`» рассматриваются как буквы, и сортировка осуществляется строго по символьному коду байтов.

Для кодировки `UTF8` дополнительно поддерживается имя `ucs_basic`, определённое в стандарте SQL. Это правило сортировки равнозначно правилу `C` и производит сортировку по кодам символов Unicode.

22.2.2.2. Предопределённые правила сортировки

Если операционная система поддерживает использование нескольких локалей в одной программе (`newlocale` и связанные функции) или включена поддержка ICU, то при инициализации кластера баз данных программа `initdb` наполняет системный каталог `pg_collation` информацией обо всех локалях, которые обнаруживает в этот момент в операционной системе.

Для просмотра всех имеющихся локалей выполните запрос `SELECT * FROM pg_collation` или команду `\dOS+` в psql.

22.2.2.2.1. Правила сортировки libc

Например, операционная система может предоставлять локаль с именем `ru_RU.utf8` При этом программа `initdb` создаст правило сортировки с именем `ru_RU.utf8` для кодировки `UTF8`, в котором `LC_COLLATE` и `LC_CTYPE` будут равны `ru_RU.utf8`. Также будет создано правило сортировки с именем без метки `.utf8` в окончании. Таким образом, вы можете использовать это правило под именем `ru_RU`, которое будет компактнее и не будет зависеть от кодировки. Заметьте, что изначальный набор имён правил сортировки тем не менее зависит от платформы.

Стандартный набор правил сортировки, предоставляемый провайдером libc, сопоставляется непосредственно с локалями, установленными в операционной системе (их можно просмотреть с помощью команды locale -a). В случаях, когда у правила сортировки libc должны быть различные значения LC_COLLATE и LC_CTYPE, или когда в операционную систему после инициализации СУБД устанавливаются новые локали, создать новое правило сортировки можно с помощью команды CREATE COLLATION. Новые локали операционной системы можно также импортировать в массовом порядке, воспользовавшись функцией pg_import_system_collations().

В любой базе данных имеют значение только те правила сортировки, которые используют кодировку этой базы данных. Прочие записи в pg_collation игнорируются. Таким образом, усечённое имя правил сортировки, такое как ru_RU, может считаться уникальным внутри данной базы данных, даже если бы оно не было уникальным глобально. Использование усечённого имени сортировки рекомендуется, так как при переходе на другую кодировку базы данных придётся выполнить на одно изменение меньше. Однако следует помнить, что правила сортировки default, C и POSIX можно использовать независимо от кодировки базы данных.

В Postgres Pro предполагается, что отдельные объекты правил сортировки несовместимы, даже когда они имеют идентичные свойства. Так, например,

```
SELECT a COLLATE "C" < b COLLATE "POSIX" FROM test1;
```

выведет сообщение об ошибке, несмотря на то, что поведение правил сортировки C и POSIX идентично. По этой причине смешивать усечённые и полные имена правил сортировки не рекомендуется.

22.2.2.2.2. Правила сортировки ICU

С ICU не представляется разумным перечислять все возможные имена локалей. ICU использует для локалей определённую схему именования, но имён локалей может быть гораздо больше, чем собственно различных локалей. Программа initdb, используя API ICU, извлекает список различных локалей и наполняет начальный набор правил в базе данных. Правила сортировки провайдера ICU создаются с именами, включающими метку языка в формате BCP 47 и указание расширения «для частного использования» (-x-icu), для отличия от локалей libc.

Например, могут быть созданы такие правила сортировки:

ru-x-icu

Русское правило сортировки, стандартный вариант

ru-UA-x-icu

Русское правило сортировки для Украины, стандартный вариант

(Например, существует правило ru-RU-x-icu, но на момент написания документации оно равнозначно правилу ru-x-icu.)

und-x-icu («undefined», неопределённая)

«Корневое» правило сортировки ICU. Оно устанавливает разумный языконезависимый порядок сортировки.

Некоторые (редко используемые) кодировки не поддерживаются ICU. Когда база имеет одну из таких кодировок, записи правил сортировки ICU в pg_collation игнорируются. При попытке использовать их будет выдана ошибка с сообщением вида «правило сортировки "de-x-icu" для кодировки "WIN874" не существует».

22.2.2.3. Создание новых правил сортировки

Если стандартных и предопределённых правил сортировки недостаточно, пользователи могут создавать собственные правила сортировки, используя команду SQL CREATE COLLATION.

Стандартные и предопределённые правила сортировки находятся в схеме pg_catalog, как и все предопределённые объекты. Пользовательские правила сортировки должны создаваться в пользовательских схемах. Помимо прочего, это полезно тем, что их будет выгружать pg_dump.

22.2.2.3.1. Правила сортировки libc

Новые правила сортировки libc могут создаваться так:

```
CREATE COLLATION russian (provider = libc, locale = 'ru_RU');
```

Точные значения, которые могут допускаться в предложении `locale` в этой команде, зависят от операционной системы. В Unix-подобных системах их список выдаёт команда `locale -a`.

Так как предопределённый набор правил сортировки libc уже включает все правила сортировки, определённые в операционной системе в момент инициализации базы данных, необходимость создавать новые правила вручную обычно не возникает. Такая потребность может возникнуть, когда нужно сменить систему именования (в этом случае см. Подраздел 22.2.2.3.3) либо когда операционная система была обновлена и в ней появились новые определения локалей (в этом случае см. `pg_import_system_collations()`).

22.2.2.3.2. Правила сортировки ICU

ICU допускает видоизменения правил сортировки, не ограничивая пользователей наборами язык+страна, которые подготавливает `initdb`. Пользователи могут свободно создавать собственные объекты-правила сортировки, использующие предоставляемые средства для получения требуемых порядков сортировки. Информацию об именовании локалей ICU можно найти на страницах *https://unicode-org.github.io/icu/userguide/locale/* и *https://unicode-org.github.io/icu/userguide/collation/api.html*. Набор допустимых имён и атрибутов зависит от конкретной версии ICU.

Несколько примеров:

```
CREATE COLLATION "de-u-co-phonebk-x-icu" (provider = icu, locale = 'de-u-co-phonebk');
CREATE COLLATION "de-u-co-phonebk-x-icu" (provider = icu, locale = 'de@collation=phonebook');
```

Немецкое правило сортировки с порядком, принятым для телефонной книги

В первом примере локаль ICU выбирается по «метке языка» в формате BCP 47. Во втором примере используется традиционный принятый в ICU синтаксис имени. Первый вариант является более предпочтительным в перспективе, но он не поддерживается старыми версиями ICU.

Заметьте, что объекты-правила сортировки в среде SQL вы можете называть как угодно. В этом примере мы следуем стилю именования, который используется предопределёнными правилами, которые в свою очередь следуют BCP 47, но это не требуется для правил сортировки, определяемых пользователем.

```
CREATE COLLATION "und-u-co-emoji-x-icu" (provider = icu, locale = 'und-u-co-emoji');
CREATE COLLATION "und-u-co-emoji-x-icu" (provider = icu, locale = '@collation=emoji');
```

Корневое правило с сортировкой эмодзи, соответствующее техническому стандарту Unicode №51

Заметьте, что в традиционной системе именования локалей ICU корневая локаль выбирается пустой строкой.

```
CREATE COLLATION latn_cyrl (provider = icu, locale = 'ru-RU-u-kr-latn-cyrl');
CREATE COLLATION latn_cyrl (provider = icu, locale = 'ru@colReorder=latn-cyrl');
```

Правило сортировки, с которым латинские буквы идут перед буквами кириллицы. (По умолчанию сначала идут буквы кириллицы.)

```
CREATE COLLATION upperfirst (provider = icu, locale = 'ru-RU-u-kf-upper');
CREATE COLLATION upperfirst (provider = icu, locale = 'ru@colCaseFirst=upper');
```

Правило сортировки, с которым буквы в верхнем регистре идут перед буквами в нижнем. (По умолчанию сначала идут буквы в нижнем регистре.)

```
CREATE COLLATION special (provider = icu, locale = 'ru-RU-u-kf-upper-kr-latn-cyrl');
CREATE COLLATION special (provider = icu, locale =
'ru@colCaseFirst=upper;colReorder=latn-cyrl');
```

Правило, в котором сочетаются предыдущие свойства.

```
CREATE COLLATION numeric (provider = icu, locale = 'en-u-kn-true');
CREATE COLLATION numeric (provider = icu, locale = 'en@colNumeric=yes');
```

Числовая сортировка, с которой последовательности чисел упорядочиваются по числовому значению, например: `A-21` < `A-123` (также называется естественной сортировкой).

За подробностями обратитесь к описанию *Технического стандарта Unicode №35* и *BCP 47*. Список возможных типов сортировки (внутренняя метка `co`) можно найти в *репозитории CLDR*.

Заметьте, что хотя данная система позволяет создавать правила сортировки, которые «игнорируют регистр» или «игнорируют ударения» и тому подобное (используя ключ `ks`), Postgres Pro в данный момент не позволяет применять такие правила сортировки действительно независящим от регистра или ударения способом. Строки, которые считаются равными согласно правилу сортировки, но не равны побайтово, будут сортироваться по своим байтовым значениям.

> **Примечание**
>
> ICU по природе своей принимает в качестве имени локали практическую любую строку и сопоставляет её с наиболее подходящей локалью, которую она может предоставить, следуя процедуре выбора, описанной в её документации. Таким образом, если указание правила сортировки будет составлено с использованием характеристик, которые данная инсталляция ICU на самом деле не поддерживает, непосредственно узнать об этом нельзя. Поэтому, чтобы убедиться, что определения правил сортировки соответствует требованиям, рекомендуется проверять поведение правил на уровне приложения.

22.2.2.3.3. Копирование правил сортировки

Команда CREATE COLLATION может также создать новое правило сортировки из существующего, что может быть полезно для использования имён, независимых от операционных систем, создания имён для совместимости или использования правил сортировки ICU под более понятными именами. Например:

```
CREATE COLLATION russian FROM "ru_RU";
CREATE COLLATION french FROM "fr-x-icu";
```

22.3. Поддержка кодировок

Поддержка кодировок в Postgres Pro позволяет хранить текст в различных кодировках, включая однобайтовые кодировки, такие как входящие в семейство ISO 8859 и многобайтовые кодировки, такие как EUC (Extended Unix Code), UTF-8 и внутренний код Mule. Все поддерживаемые кодировки могут прозрачно использоваться клиентами, но некоторые не поддерживаются сервером (в качестве серверной кодировки). Кодировка по умолчанию выбирается при инициализации кластера базы данных Postgres Pro при помощи `initdb`. Она может быть переопределена при создании базы данных, что позволяет иметь несколько баз данных с разными кодировками.

Важным ограничением, однако, является то, что кодировка каждой базы данных должна быть совместима с параметрами локали базы данных `LC_CTYPE` (классификация символов) и `LC_COLLATE` (порядок сортировки строк). Для локали `C` или `POSIX` подойдёт любой набор символов, но для других локалей, предоставляемых библиотекой libc, есть только один набор символов, который будет работать правильно. (Однако в среде Windows кодировка UTF-8 может использоваться с любой локалью.) Если у вас включена поддержка ICU, локали, предоставляемые библиотекой ICU, можно использовать с большинством (но не всеми) кодировками на стороне сервера.

22.3.1. Поддерживаемые кодировки

Таблица 22.1 показывает кодировки, доступные для использования в Postgres Pro.

Таблица 22.1. Кодировки Postgres Pro

Имя	Описание	Язык	Поддержка на сервере	ICU?	Байтов на символ	Псевдонимы
BIG5	Big Five	Традиционные китайские иероглифы	Нет	Нет	1-2	WIN950, Windows950
EUC_CN	Extended UNIX Code-CN	Упрощённые китайские иероглифы	Да	Да	1-3	
EUC_JP	Extended UNIX Code-JP	Японский	Да	Да	1-3	
EUC_JIS_2004	Extended UNIX Code-JP, JIS X 0213	Японский	Да	Нет	1-3	
EUC_KR	Extended UNIX Code-KR	Корейский	Да	Да	1-3	
EUC_TW	Extended UNIX Code-TW	Традиционные китайские иероглифы, тайваньский	Да	Да	1-3	
GB18030	Национальный стандарт	Китайский	Нет	Нет	1-4	
GBK	Расширенный национальный стандарт	Упрощённые китайские иероглифы	Нет	Нет	1-2	WIN936, Windows936
ISO_8859_5	ISO 8859-5, ECMA 113	Латинский/ Кириллица	Да	Да	1	
ISO_8859_6	ISO 8859-6, ECMA 114	Латинский/ Арабский	Да	Да	1	
ISO_8859_7	ISO 8859-7, ECMA 118	Латинский/ Греческий	Да	Да	1	
ISO_8859_8	ISO 8859-8, ECMA 121	Латинский/ Иврит	Да	Да	1	
JOHAB	JOHAB	Корейский (Хангыль)	Нет	Нет	1-3	
KOI8R	KOI8-R	Кириллица (Русский)	Да	Да	1	KOI8
KOI8U	KOI8-U	Кириллица (Украинский)	Да	Да	1	
LATIN1	ISO 8859-1, ECMA 94	Западно европейские	Да	Да	1	ISO88591
LATIN2	ISO 8859-2, ECMA 94	Центрально европейские	Да	Да	1	ISO88592
LATIN3	ISO 8859-3, ECMA 94	Южно европейские	Да	Да	1	ISO88593

Имя	Описание	Язык	Поддержка на сервере	ICU?	Байтов на символ	Псевдонимы
LATIN4	ISO 8859-4, ECMA 94	Северо европейские	Да	Да	1	ISO88594
LATIN5	ISO 8859-9, ECMA 128	Турецкий	Да	Да	1	ISO88599
LATIN6	ISO 8859-10, ECMA 144	Скандинавский	Да	Да	1	ISO885910
LATIN7	ISO 8859-13	Балтийские	Да	Да	1	ISO885913
LATIN8	ISO 8859-14	Кельтские	Да	Да	1	ISO885914
LATIN9	ISO 8859-15	LATIN1 с европейскими языками и диалектами	Да	Да	1	ISO885915
LATIN10	ISO 8859-16, ASRO SR 14111	Румынский	Да	Нет	1	ISO885916
MULE_ INTERNAL	Внутренний код Mule	Мультиязычный редактор Emacs	Да	Нет	1-4	
SJIS	Shift JIS	Японский	Нет	Нет	1-2	Mskanji, ShiftJIS, WIN932, Windows932
SHIFT_JIS_ 2004	Shift JIS, JIS X 0213	Японский	Нет	Нет	1-2	
SQL_ASCII	не указан (см. текст)	*any*	Да	Нет	1	
UHC	Унифицированный код Хангыль	Корейский	Нет	Нет	1-2	WIN949, Windows949
UTF8	Unicode, 8-bit	*все*	Да	Да	1-4	Unicode
WIN866	Windows CP866	Кириллица	Да	Да	1	ALT
WIN874	Windows CP874	Тайский	Да	Нет	1	
WIN1250	Windows CP1250	Центрально европейские	Да	Да	1	
WIN1251	Windows CP1251	Кириллица	Да	Да	1	WIN
WIN1252	Windows CP1252	Западно европейские	Да	Да	1	
WIN1253	Windows CP1253	Греческий	Да	Да	1	
WIN1254	Windows CP1254	Турецкий	Да	Да	1	
WIN1255	Windows CP1255	Иврит	Да	Да	1	

Имя	Описание	Язык	Поддержка на сервере	ICU?	Байтов на символ	Псевдонимы
WIN1256	Windows CP1256	Арабский	Да	Да	1	
WIN1257	Windows CP1257	Балтийские	Да	Да	1	
WIN1258	Windows CP1258	Вьетнамский	Да	Да	1	ABC, TCVN, TCVN5712, VSCII

Не все клиентские API поддерживают все перечисленные кодировки. Например, драйвер интерфейса JDBC Postgres Pro не поддерживает MULE_INTERNAL, LATIN6, LATIN8 и LATIN10.

Поведение кодировки SQL_ASCII существенно отличается от других. Когда набором символов сервера является SQL_ASCII, сервер интерпретирует значения от 0 до 127 байт согласно кодировке ASCII, тогда как значения от 128 до 255 воспринимаются как незначимые. Перекодировка не будет выполнена при выборе SQL_ASCII. Таким образом, этот вариант является не столько объявлением того, что используется определённая кодировка, сколько объявлением того, что кодировка игнорируется. В большинстве случаев, если вы работаете с любыми данными, отличными от ASCII, не стоит использовать SQL_ASCII, так как Postgres Pro не сможет преобразовать или проверить символы, отличные от ASCII.

22.3.2. Настройка кодировки

initdb определяет кодировку по умолчанию для кластера Postgres Pro. Например,

```
initdb -E EUC_JP
```

настраивает кодировку по умолчанию на EUC_JP (Расширенная система кодирования для японского языка). Можно использовать --encoding вместо -E в случае предпочтения более длинных имён параметров. Если параметр -E или --encoding не задан, initdb пытается определить подходящую кодировку в зависимости от указанной или заданной по умолчанию локали.

При создании базы данных можно указать кодировку, отличную от заданной по умолчанию, если эта кодировка совместима с выбранной локалью:

```
createdb -E EUC_KR -T template0 --lc-collate=ko_KR.euckr --lc-ctype=ko_KR.euckr korean
```

Это создаст базу данных с именем korean, которая использует кодировку EUC_KR и локаль ko_KR. Также, получить желаемый результат можно с помощью данной SQL-команды:

```
CREATE DATABASE korean WITH ENCODING 'EUC_KR' LC_COLLATE='ko_KR.euckr'
 LC_CTYPE='ko_KR.euckr' TEMPLATE=template0;
```

Заметьте, что приведённые выше команды задают копирование базы данных template0. При копировании любой другой базы данных, параметры локали и кодировку исходной базы изменить нельзя, так как это может привести к искажению данных. Более подробное описание приведено в Разделе 21.3.

Кодировка базы данных хранится в системном каталоге pg_database. Её можно увидеть при помощи параметра psql -l или команды \l.

```
$ psql -l
                                        List of databases
   Name    |  Owner   | Encoding  |  Collation  |    Ctype    |          Access
 Privileges
-----------+----------+-----------+-------------+-------------
+-----------------------------------
```

```
 clocaledb | hlinnaka | SQL_ASCII | C           | C           |
 englishdb | hlinnaka | UTF8      | en_GB.UTF8  | en_GB.UTF8  |
 japanese  | hlinnaka | UTF8      | ja_JP.UTF8  | ja_JP.UTF8  |
 korean    | hlinnaka | EUC_KR    | ko_KR.euckr | ko_KR.euckr |
 postgres  | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  |
 template0 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  | {=c/
hlinnaka,hlinnaka=CTc/hlinnaka}
 template1 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  | {=c/
hlinnaka,hlinnaka=CTc/hlinnaka}
(7 rows)
```

> **Важно**
>
> На большинстве современных операционных систем Postgres Pro может определить, какая кодировка подразумевается параметром `LC_CTYPE`, что обеспечит использование только соответствующей кодировки базы данных. На более старых системах необходимо самостоятельно следить за тем, чтобы использовалась кодировка, соответствующая выбранной языковой среде. Ошибка в этой области, скорее всего, приведёт к странному поведению зависимых от локали операций, таких как сортировка.
>
> Postgres Pro позволит суперпользователям создавать базы данных с кодировкой `SQL_ASCII`, даже когда значение `LC_CTYPE` не установлено в `C` или `POSIX`. Как было сказано выше, `SQL_ASCII` не гарантирует, что данные, хранящиеся в базе, имеют определённую кодировку, и таким образом, этот выбор чреват сбоями, связанными с локалью. Использование данной комбинации устарело и, возможно, будет полностью запрещено.

22.3.3. Автоматическая перекодировка между сервером и клиентом

Postgres Pro поддерживает автоматическую перекодировку между сервером и клиентом для определённых комбинаций кодировок. Информация, касающаяся перекодировки, хранится в системном каталоге `pg_conversion`. Postgres Pro включает в себя некоторые предопределённые кодировки, как показано в Таблице 22.2. Есть возможность создать новую перекодировку при помощи SQL-команды `CREATE CONVERSION`.

Таблица 22.2. Клиент-серверные перекодировки наборов символов

Серверная кодировка	Доступные клиентские кодировки
`BIG5`	*не поддерживается как серверная кодировка*
`EUC_CN`	*`EUC_CN`*, `MULE_INTERNAL`, `UTF8`
`EUC_JP`	*`EUC_JP`*, `MULE_INTERNAL`, `SJIS`, `UTF8`
`EUC_JIS_2004`	*`EUC_JIS_2004`*, `SHIFT_JIS_2004`, `UTF8`
`EUC_KR`	*`EUC_KR`*, `MULE_INTERNAL`, `UTF8`
`EUC_TW`	*`EUC_TW`*, `BIG5`, `MULE_INTERNAL`, `UTF8`
`GB18030`	*не поддерживается как серверная кодировка*
`GBK`	*не поддерживается как серверная кодировка*
`ISO_8859_5`	*`ISO_8859_5`*, `KOI8R`, `MULE_INTERNAL`, `UTF8`, `WIN866`, `WIN1251`
`ISO_8859_6`	*`ISO_8859_6`*, `UTF8`
`ISO_8859_7`	*`ISO_8859_7`*, `UTF8`
`ISO_8859_8`	*`ISO_8859_8`*, `UTF8`

Серверная кодировка	Доступные клиентские кодировки
JOHAB	*не поддерживается как серверная кодировка*
KOI8R	*KOI8R*, ISO_8859_5 , MULE_INTERNAL , UTF8, WIN866, WIN1251
KOI8U	*KOI8U*, UTF8
LATIN1	*LATIN1*, MULE_INTERNAL , UTF8
LATIN2	*LATIN2*, MULE_INTERNAL , UTF8, WIN1250
LATIN3	*LATIN3*, MULE_INTERNAL , UTF8
LATIN4	*LATIN4*, MULE_INTERNAL , UTF8
LATIN5	*LATIN5*, UTF8
LATIN6	*LATIN6*, UTF8
LATIN7	*LATIN7*, UTF8
LATIN8	*LATIN8*, UTF8
LATIN9	*LATIN9*, UTF8
LATIN10	*LATIN10*, UTF8
MULE_INTERNAL	*MULE_INTERNAL*, BIG5, EUC_CN , EUC_JP , EUC_KR , EUC_TW , ISO_8859_5 , KOI8R, LATIN1 to LATIN4, SJIS, WIN866, WIN1250, WIN1251
SJIS	*не поддерживается как серверная кодировка*
SHIFT_JIS_2004	*не поддерживается как серверная кодировка*
SQL_ASCII	*любая (перекодировка не будет выполнена)*
UHC	*не поддерживается как серверная кодировка*
UTF8	*все поддерживаемые кодировки*
WIN866	*WIN866*, ISO_8859_5 , KOI8R, MULE_INTERNAL , UTF8, WIN1251
WIN874	*WIN874*, UTF8
WIN1250	*WIN1250*, LATIN2, MULE_INTERNAL , UTF8
WIN1251	*WIN1251*, ISO_8859_5 , KOI8R, MULE_INTERNAL , UTF8, WIN866
WIN1252	*WIN1252*, UTF8
WIN1253	*WIN1253*, UTF8
WIN1254	*WIN1254*, UTF8
WIN1255	*WIN1255*, UTF8
WIN1256	*WIN1256*, UTF8
WIN1257	*WIN1257*, UTF8
WIN1258	*WIN1258*, UTF8

Чтобы включить автоматическую перекодировку символов, необходимо сообщить Postgres Pro кодировку, которую вы хотели бы использовать на стороне клиента. Это можно выполнить несколькими способами:

- Использование команды `\encoding` в psql. `\encoding` позволяет оперативно изменять клиентскую кодировку. Например, чтобы изменить кодировку на `SJIS`, введите:

  ```
  \encoding SJIS
  ```

- libpq (Раздел 31.10) имеет функции, для управления клиентской кодировкой.

- Использование `SET client_encoding TO`. Клиентская кодировка устанавливается следующей SQL-командой:

  ```
  SET CLIENT_ENCODING TO 'value';
  ```

  Также, для этой цели можно использовать стандартный синтаксис SQL `SET NAMES`:

  ```
  SET NAMES 'value';
  ```

  Получить текущую клиентскую кодировку:

  ```
  SHOW client_encoding;
  ```

  Вернуть кодировку по умолчанию:

  ```
  RESET client_encoding;
  ```

- Использование `PGCLIENTENCODING`. Если установлена переменная окружения `PGCLIENTENCODING`, то эта клиентская кодировка выбирается автоматически при подключении к серверу. (В дальнейшем это может быть переопределено при помощи любого из методов, указанных выше.)
- Использование переменной конфигурации client_encoding. Если задана переменная `client_encoding`, указанная клиентская кодировка выбирается автоматически при подключении к серверу. (В дальнейшем это может быть переопределено при помощи любого из методов, указанных выше.)

Если перекодировка определённого символа невозможна (предположим, выбраны `EUC_JP` для сервера и `LATIN1` для клиента, и передаются некоторые японские иероглифы, не представленные в `LATIN1`), возникает ошибка.

Если клиентская кодировка определена как `SQL_ASCII`, перекодировка отключается вне зависимости от кодировки сервера. Что же касается сервера, не стоит использовать `SQL_ASCII`, если только вы не работаете с данными, которые полностью соответствуют ASCII.

22.3.4. Дополнительные источники информации

Рекомендуемые источники для начала изучения различных видов систем кодирования.

Обработка информации на китайском, японском, корейском & вьетнамском языках.

Содержит подробные объяснения по `EUC_JP`, `EUC_CN`, `EUC_KR`, `EUC_TW`.

http://www.unicode.org/

Сайт Unicode Consortium.

RFC 3629

UTF-8 (формат преобразования 8-битного UCS/Unicode) определён здесь.

Глава 23. Регламентные задачи обслуживания базы данных

Как и в любой СУБД, в Postgres Pro для достижения оптимальной производительности нужно регулярно выполнять определённые процедуры. Задачи, которые рассматриваются в этой главе, являются *обязательными*, но они по природе своей повторяющиеся и легко поддаются автоматизации с использованием стандартных средств, таких как задания cron или Планировщика задач в Windows. Создание соответствующих заданий и контроль над их успешным выполнением входят в обязанности администратора базы данных.

Одной из очевидных задач обслуживания СУБД является регулярное создание резервных копий данных. При отсутствии свежей резервной копии у вас не будет шанса восстановить систему после катастрофы (сбой диска, пожар, удаление важной таблицы по ошибке и т. д.). Механизмы резервного копирования и восстановления в Postgres Pro детально рассматриваются в Главе 24.

Другое важное направление обслуживания СУБД — периодическая «очистка» базы данных. Эта операция рассматривается в Разделе 23.1. С ней тесно связано обновление статистики, которая будет использоваться планировщиком запросов; оно рассматривается в Подразделе 23.1.3.

Ещё одной задачей, требующей периодического выполнения, является управление файлами журнала. Она рассматривается в Разделе 23.3.

Для контроля состояния базы данных и для отслеживания нестандартных ситуаций можно использовать *check_postgres*. Скрипт check_postgres можно интегрировать с Nagios и MRTG, однако он может работать и самостоятельно.

По сравнению с некоторыми другими СУБД Postgres Pro неприхотлив в обслуживании. Тем не менее должное внимание к вышеперечисленным задачам будет значительно способствовать комфортной и производительной работе с СУБД.

23.1. Регламентная очистка

Базы данных Postgres Pro требуют периодического проведения процедуры обслуживания, которая называется *очисткой*. Во многих случаях очистку достаточно выполнять с помощью *демона автоочистки*, который описан в Подразделе 23.1.6. Возможно, в вашей ситуации для получения оптимальных результатов потребуется настроить описанные там же параметры автоочистки. Некоторые администраторы СУБД могут дополнить или заменить действие этого демона командами `VACUUM` (обычно они выполняются по расписанию в заданиях cron или Планировщика задач). Чтобы правильно организовать очистку вручную, необходимо понимать темы, которые будут рассмотрены в следующих подразделах. Администраторы, которые полагаются на автоочистку, возможно, всё же захотят просмотреть этот материал, чтобы лучше понимать и настраивать эту процедуру.

23.1.1. Основные принципы очистки

Команды VACUUM в Postgres Pro должны обрабатывать каждую таблицу по следующим причинам:

1. Для высвобождения или повторного использования дискового пространства, занятого изменёнными или удалёнными строками.
2. Для обновления статистики по данным, используемой планировщиком запросов Postgres Pro.
3. Для обновления карты видимости, которая ускоряет сканирование только индекса.
4. Для предотвращения потери очень старых данных из-за зацикливания идентификаторов *транзакций* или *мультитранзакций*.

Разные причины диктуют выполнение действий `VACUUM` с разной частотой и в разном объёме, как рассматривается в следующих подразделах.

Существует два варианта VACUUM: обычный VACUUM и VACUUM FULL. Команда VACUUM FULL может высвободить больше дискового пространства, однако работает медленнее. Кроме того, обычная команда VACUUM может выполняться параллельно с использованием производственной базы данных. (При этом такие команды как SELECT, INSERT, UPDATE и DELETE будут выполняться нормально, хотя нельзя будет изменить определение таблицы командами типа ALTER TABLE.) Команда VACUUM FULL требует блокировки обрабатываемой таблицы в режиме ACCESS EXCLUSIVE и поэтому не может выполняться параллельно с другими операциями с этой таблицей. По этой причине администраторы, как правило, должны стараться использовать обычную команду VACUUM и избегать VACUUM FULL.

Команда VACUUM порождает существенный объём трафика ввода/вывода, который может стать причиной низкой производительности в других активных сеансах. Это влияние фоновой очистки можно регулировать, настраивая параметры конфигурации (см. Подраздел 18.4.4).

23.1.2. Высвобождение дискового пространства

В Postgres Pro команды UPDATE или DELETE не вызывают немедленного удаления старой версии изменяемых строк. Этот подход необходим для реализации эффективного многоверсионного управления конкурентным доступом (MVCC, см. Главу 13): версия строки не должна удаляться до тех пор, пока она остаётся потенциально видимой для других транзакций. Однако в конце концов устаревшая или удалённая версия строки оказывается не нужна ни одной из транзакций. После этого занимаемое ей место должно быть освобождено и может быть отдано новым строкам, во избежание неограниченного роста потребности в дисковом пространстве. Это происходит при выполнении команды VACUUM.

Обычная форма VACUUM удаляет неиспользуемые версии строк в таблицах и индексах и помечает пространство свободным для дальнейшего использования. Однако это дисковое пространство не возвращается операционной системе, кроме особого случая, когда полностью освобождаются одна или несколько страниц в конце таблицы и можно легко получить исключительную блокировку таблицы. Команда VACUUM FULL, напротив, кардинально сжимает таблицы, записывая абсолютно новую версию файла таблицы без неиспользуемого пространства. Это минимизирует размер таблицы, однако может занять много времени. Кроме того, для этого требуется больше места на диске для записи новой копии таблицы до завершения операции.

Обычно цель регулярной очистки — выполнять простую очистку (VACUUM) достаточно часто, чтобы не возникала необходимость в VACUUM FULL. Демон автоочистки пытается работать в этом режиме, и на самом деле он сам никогда не выполняет VACUUM FULL. Основная идея такого подхода не в том, чтобы минимизировать размер таблиц, а в том, чтобы поддерживать использование дискового пространства на стабильном уровне: каждая таблица занимает объём, равный её минимальному размеру, плюс объём, который был занят между процедурами очистки. Хотя с помощью VACUUM FULL можно сжать таблицу до минимума и возвратить дисковое пространство операционной системе, большого смысла в этом нет, если в будущем таблица так же вырастет снова. Следовательно, для активно изменяемых таблиц лучше с умеренной частотой выполнять VACUUM, чем очень редко выполнять VACUUM FULL.

Некоторые администраторы предпочитают планировать очистку БД самостоятельно, например, проводя все работы ночью в период низкой загрузки. Однако очистка только по фиксированному расписанию плоха тем, что при резком скачке интенсивности изменений таблица может разрастить настолько, что для высвобождения пространства действительно понадобится выполнить VACUUM FULL. Использование демона автоочистки снимает эту проблему, поскольку он планирует очистку динамически, отслеживая интенсивность изменений. Полностью отключать этот демон может иметь смысл, только если вы имеете дело с предельно предсказуемой загрузкой. Возможен и компромиссный вариант — настроить параметры демона автоочистки так, чтобы он реагировал только на необычайно высокую интенсивность изменений и мог удержать ситуацию под контролем, в то время как команды VACUUM, запускаемые по расписанию, будут выполнять основную работу в периоды нормальной загрузки.

Если же автоочистка не применяется, обычно планируется выполнение VACUUM для всей базы данных раз в сутки в период низкой активности, и в случае необходимости оно дополняется

более частой очисткой интенсивно изменяемых таблиц. (В некоторых ситуациях, когда изменения производятся крайне интенсивно, самые востребованные таблицы могут очищаться раз в несколько минут.) Если в вашем кластере несколько баз данных, не забывайте выполнять VACUUM для каждой из них; при этом может быть полезна программа vacuumdb.

> **Подсказка**
>
> Результат обычного VACUUM может быть неудовлетворительным, когда вследствие массового изменения или удаления строк в таблице оказывается много мёртвых версий строк. Если у вас есть такая таблица и вам нужно освободить лишнее пространство, которое она занимает, используйте команду VACUUM FULL или, в качестве альтернативы, CLUSTER или один из вариантов ALTER TABLE, выполняющий перезапись таблицы. Эти команды записывают абсолютно новую копию таблицы и строят для неё индексы. Все эти варианты требуют блокировки в режиме ACCESS EXCLUSIVE. Заметьте, что они также на время требуют дополнительного пространства на диске в объёме, приблизительно равном размеру таблицы, поскольку старые копии таблицы и индексов нельзя удалить до завершения создания новых копий.

> **Подсказка**
>
> Если у вас есть таблица, всё содержимое которой периодически нужно удалять, имеет смысл делать это, выполняя только TRUNCATE, а не DELETE и затем VACUUM. TRUNCATE немедленно удаляет всё содержимое таблицы, не требуя последующей очистки (VACUUM или VACUUM FULL) для высвобождения неиспользуемого дискового пространства. Недостатком такого подхода является нарушение строгой семантики MVCC.

23.1.3. Обновление статистики планировщика

Планировщик запросов в Postgres Pro, выбирая эффективные планы запросов, полагается на статистическую информацию о содержимом таблиц. Эта статистика собирается командой ANALYZE, которая может вызываться сама по себе или как дополнительное действие команды VACUUM. Статистика должна быть достаточно точной, так как в противном случае неудачно выбранные планы запросов могут снизить производительность базы данных.

Демон автоочистки, если он включён, будет автоматически выполнять ANALYZE после существенных изменений содержимого таблицы. Однако администраторы могут предпочесть выполнение ANALYZE вручную, в частности, если известно, что производимые в таблице изменения не повлияют на статистику по «интересным» столбцам. Демон же планирует выполнение ANALYZE в зависимости только от количества вставленных или изменённых строк; он не знает, приведут ли они к значимым изменениям статистики.

Изменение кортежей в дочерних таблицах и секциях не считается поводом для запуска анализа родительской таблицы. Если родительская таблица пуста или изменяется редко, процесс автоочистки может никогда не затронуть её и статистика иерархии наследования собираться не будет. Поэтому необходимо выполнять ANALYZE для родительской таблицы вручную, чтобы поддерживать статистику в актуальном состоянии.

Как и процедура очистки для высвобождения пространства, частое обновление статистики полезнее для интенсивно изменяемых таблиц, нежели для тех таблиц, которые изменяются редко. Однако даже в случае часто изменяемой таблицы обновление статистики может не требоваться, если статистическое распределение данных меняется слабо. Как правило, достаточно оценить, насколько меняются максимальное и минимальное значения в столбцах таблицы. Например, максимальное значение в столбце timestamp, хранящем время изменения строки, будет постоянно увеличиваться по мере добавления и изменения строк; для такого столбца может потребоваться более частое обновление статистики, чем, к примеру, для столбца, содержащего адреса страниц

(URL), которые запрашивались с сайта. Столбец с URL-адресами может меняться столь же часто, однако статистическое распределение его значений, вероятно, будет изменяться относительно медленно.

Команду ANALYZE можно выполнять для отдельных таблиц и даже просто для отдельных столбцов таблицы, поэтому, если того требует приложение, одни статистические данные можно обновлять чаще, чем другие. Однако на практике обычно лучше просто анализировать всю базу данных, поскольку это быстрая операция, так как ANALYZE читает не каждую отдельную строку, а статистически случайную выборку строк таблицы.

Подсказка

Хотя индивидуальная настройка частоты ANALYZE для отдельных столбцов может быть не очень полезной, смысл может иметь настройка детализации статистики, собираемой командой ANALYZE. Для столбцов, которые часто используются в предложениях WHERE, и имеют очень неравномерное распределение данных, может потребоваться более детальная, по сравнению с другими столбцами, гистограмма данных. В таких случаях можно воспользоваться командой ALTER TABLE SET STATISTICS или изменить значение по умолчанию параметра уровня БД default_statistics_target.

Кроме того, по умолчанию информация об избирательности функций ограничена. Однако если вы создаёте индекс по выражению с вызовом функции, об этой функции будет собрана полезная статистическая информация, которая может значительно улучшить планы запросов, в которых используется данный индекс.

Подсказка

Демон автоочистки не выполняет команды ANALYZE для сторонних таблиц, поскольку он не знает, как часто это следует делать. Если для получения качественных планов вашим запросам необходима статистика по сторонним таблицам, будет хорошей идеей дополнительно запускать ANALYZE для них по подходящему расписанию.

Подсказка

Демон автоочистки не выполняет команды ANALYZE для секционированных таблиц. В иерархии наследования анализ родительской таблицы проводится, только если меняются данные в ней самой — изменения в дочерней таблице не считаются поводом для автоанализа родительской таблицы. Если для получения качественных планов вашим запросам необходима статистика по родительским таблицам, нужно периодически выполнять ANALYZE вручную, чтобы их статистика была актуальной.

23.1.4. Обновление карты видимости

Процедура очистки поддерживает карты видимости для каждой таблицы, позволяющие определить, в каких страницах есть только записи, заведомо видимые для всех активных транзакций (и всех будущих транзакций, пока страница не будет изменена). Это имеет два применения. Во-первых, сам процесс очистки может пропускать такие страницы при следующем запуске, поскольку на этих страницах вычищать нечего.

Во-вторых, с такими картами Postgres Pro может выдавать результаты некоторых запросов, используя только индекс, не обращаясь к данным таблицы. Так как индексы Postgres Pro не содержат информацию о видимости записей, при обычном сканировании по индексу необходимо извлечь соответствующую запись из таблицы и проверить её видимость для текущей транзакции.

Поэтому при *сканировании только индекса*, наоборот, сначала проверяется карта видимости. Если известно, что все записи на странице видимы, то выборку из таблицы можно пропустить. Это наиболее полезно с большими наборах данных, когда благодаря карте видимости можно оптимизировать чтение с диска. Карта видимости значительно меньше таблицы, поэтому она легко помещается в кеш, даже когда объём самих страниц очень велик.

23.1.5. Предотвращение ошибок из-за зацикливания счётчика транзакций

В Postgres Pro семантика транзакций MVCC зависит от возможности сравнения номеров идентификаторов транзакций (XID): версия строки, у которой XID добавившей её транзакции больше, чем XID текущей транзакции, относится «к будущему» и не должна быть видна в текущей транзакции. Однако поскольку идентификаторы транзакций имеют ограниченный размер (32 бита), кластер, работающий долгое время (более 4 миллиардов транзакций) столкнётся с *зацикливанием идентификаторов транзакций*: счётчик XID прокрутится до нуля, и внезапно транзакции, которые относились к прошлому, окажутся в будущем — это означает, что их результаты станут невидимыми. Одним словом, это катастрофическая потеря данных. (На самом деле данные никуда не пропадают, однако если вы не можете их получить, то это слабое утешение.) Для того чтобы этого избежать, необходимо выполнять очистку для каждой таблицы в каждой базе данных как минимум единожды на два миллиардов транзакций.

Периодическое выполнение очистки решает эту проблему, потому что процедура `VACUUM` помечает строки как *замороженные*, указывая, что они были вставлены транзакцией, зафиксированной достаточно давно, так что эффект добавляющей транзакции с точки зрения MVCC определённо будет виден во всех текущих и будущих транзакциях. Обычные значения XID сравниваются по модулю 2^{32}. Это означает, что для каждого обычного XID существуют два миллиарда значений XID, которые «старше» него, и два миллиарда значений, которые «младше» него; другими словами, пространство значений XID циклично и не имеет конечной точки. Следовательно, как только создаётся версия строки с обычным XID, для следующих двух миллиардов транзакций эта версия строки оказывается «в прошлом», неважно о каком значении обычного XID идет речь. Если после двух миллиардов транзакций эта версия строки всё ещё существует, она внезапно окажется в будущем. Для того чтобы это предотвратить, в какой-то момент значение XID для старых версий строк должно быть заменено на `FrozenTransactionId` (заморожено) до того, как будет достигнута граница в два миллиарда транзакций. После получения этого особенного XID для всех обычных транзакций эти версии строк будут относиться «к прошлому», независимо от зацикливания, и, таким образом, эти версии строк будут действительны до момента их удаления, когда бы это ни произошло.

> **Примечание**
>
> В версиях PostgreSQL до 9.4 замораживание было реализовано как замена XID добавления строки специальным идентификатором `FrozenTransactionId`, который можно было увидеть в системной колонке `xmin` данной строки. В новых версиях просто устанавливается битовый флаг, а исходный `xmin` строки сохраняется для возможного расследования в будущем. Однако строки с `xmin`, равным `FrozenTransactionId` (2), можно по-прежнему встретить в базах данных, обновлённых (с применением pg_upgrade) с версий до 9.4.
>
> Также системные каталоги могут содержать строки со значением `xmin`, равным `BootstrapTransactionId` (1), показывающим, что они были вставлены на первом этапе initdb. Как и `FrozenTransactionId`, этот специальный XID считается более старым, чем любой обычный XID.

Параметр vacuum_freeze_min_age определяет, насколько старым должен стать XID, чтобы строки с таким XID были заморожены. Увеличение его значения помогает избежать ненужной работы, если строки, которые могли бы быть заморожены в ближайшее время, будут изменены ещё раз, а уменьшение приводит к увеличению количества транзакций, которые могут выполниться, прежде чем потребуется очередная очистка таблицы.

VACUUM определяет, какие страницы таблицы нужно сканировать, анализируя карту видимости. Обычно при этой операции пропускаются страницы, в которых нет мёртвых версий строк, даже если в них могут быть версии строк со старыми XID. Таким образом, обычная команда VACUUM не будет всегда замораживать все версии строк, имеющиеся в таблице. Периодически VACUUM будет также производить *агрессивную очистку*, пропуская только те страницы, которые не содержат ни мёртвых строк, ни незамороженных значений XID или MXID. Когда VACUUM будет делать это, зависит от параметра vacuum_freeze_table_age: полностью видимые, но не полностью замороженные страницы будут сканироваться, если число транзакций, прошедших со времени последнего такого сканирования, оказывается больше чем vacuum_freeze_table_age минус vacuum_freeze_min_age. Если vacuum_freeze_table_age равно 0, VACUUM будет применять эту более агрессивную стратегию при каждом сканировании.

Максимальное время, в течение которого таблица может обходиться без очистки, составляет два миллиарда транзакций минус значение vacuum_freeze_min_age с момента последней агрессивной очистки. Если бы таблица не подвергалась очистке дольше, была бы возможна потеря данных. Чтобы гарантировать, что это не произойдёт, для любой таблицы, которая может содержать значения XID старше, чем возраст, указанный в конфигурационном параметре autovacuum_freeze_max_age, вызывается автоочистка. (Это случится, даже если автоочистка отключена.)

Это означает, что если очистка таблицы не вызывается другим способом, то автоочистка для неё будет вызываться приблизительно через каждые autovacuum_freeze_max_age минус vacuum_freeze_min_age транзакций. Для таблиц, очищаемых регулярно для высвобождения пространства, это неактуально. В то же время статичные таблицы (включая таблицы, в которых данные вставляются, но не изменяются и не удаляются) не нуждаются в очистке для высвобождения пространства, поэтому для очень больших статичных таблиц имеет смысл увеличить интервал между вынужденными запусками автоочистки. Очевидно, это можно сделать, либо увеличив autovacuum_freeze_max_age, либо уменьшив vacuum_freeze_min_age.

Фактический максимум для vacuum_freeze_table_age составляет 0.95 * autovacuum_freeze_max_age; большее значение будет ограничено этим пределом. Значение, превышающее autovacuum_freeze_max_age, не имело бы смысла, поскольку по достижении этого значения в любом случае вызывалась бы автоочистка для предотвращения зацикливания, а коэффициент 0.95 оставляет немного времени для того, чтобы запустить команду VACUUM вручную до того, как это произойдёт. Как правило, установленное значение vacuum_freeze_table_age должно быть несколько меньше autovacuum_freeze_max_age, чтобы оставленный промежуток был достаточен для выполнения в этом окне VACUUM по расписанию или автоочистки, управляемой обычной активностью операций удаления и изменения. Если это значение будет слишком близким к максимуму, автоочистка для предотвращения зацикливания будет выполняться, даже если таблица только что была очищена для высвобождения пространства, в то же время при небольшом значении будет чаще производиться агрессивная очистка.

Единственный минус увеличения autovacuum_freeze_max_age (и vacuum_freeze_table_age с ним) заключается в том, что подкаталоги pg_xact и pg_commit_ts в кластере баз данных будут занимать больше места, поскольку в них нужно будет хранить статус и (при включённом track_commit_timestamp) время фиксации всех транзакций вплоть до горизонта autovacuum_freeze_max_age. Для статуса фиксации используется по два бита на транзакцию, поэтому если в autovacuum_freeze_max_age установлено максимально допустимое значение в два миллиарда, то размер pg_xact может составить примерно половину гигабайта, а pg_commit_ts примерно 20 ГБ. Если по сравнению с объёмом вашей базы данных этот объём незначителен, тогда рекомендуется установить для autovacuum_freeze_max_age максимально допустимое значение. В противном случае установите значение этого параметра в зависимости от объёма, который вы готовы выделить для pg_xact и pg_commit_ts. (Значению по умолчанию, 200 миллионам транзакций, соответствует приблизительно 50 МБ в pg_xact и около 2 ГБ в pg_commit_ts.)

Уменьшение значения vacuum_freeze_min_age, с другой стороны, чревато тем, что команда VACUUM может выполнять бесполезную работу: замораживание версии строки — пустая трата времени, если эта строка будет вскоре изменена (и в результате получит новый XID). Поэтому значение

этого параметра должно быть достаточно большим для того, чтобы строки не замораживались, пока их последующее изменение не станет маловероятным.

Для отслеживания возраста самых старых значений XID в базе данных команда VACUUM сохраняет статистику по XID в системных таблицах pg_class и pg_database. В частности, столбец relfrozenxid в записи для определённой таблицы в pg_class содержит граничное значение XID, с которым в последний раз выполнялась агрессивная очистка (VACUUM) этой таблицы. Все строки, добавленные транзакциями с более ранними XID, гарантированно будут заморожены. Аналогично столбец datfrozenxid в записи для базы данных в pg_database представляет нижнюю границу обычных значений XID, встречающихся в этой базе — он просто хранит минимальное из всех значений relfrozenxid для таблиц этой базы. Эту информацию удобно получать с помощью таких запросов:

```
SELECT c.oid::regclass as table_name,
       greatest(age(c.relfrozenxid),age(t.relfrozenxid)) as age
FROM pg_class c
LEFT JOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.relkind IN ('r', 'm');

SELECT datname, age(datfrozenxid) FROM pg_database;
```

Столбец age показывает количество транзакций от граничного значения XID до XID текущей транзакции.

Обычно VACUUM сканирует только те страницы, которые изменялись после последней очистки, однако relfrozenxid может увеличиться только при сканировании всех страниц таблицы, включая те, что могут содержать незамороженные XID. Это происходит, когда возраст relfrozenxid превышает vacuum_freeze_table_age транзакций, когда VACUUM вызывается с указанием FREEZE, или когда оказывается, что очистку для удаления мёртвых версий строк нужно провести во всех ещё не замороженных страницах. Когда VACUUM сканирует в таблице каждую ещё не полностью замороженную страницу, значение age(relfrozenxid) в результате должно стать немного больше, чем установленное значение vacuum_freeze_min_age (больше на число транзакций, начатых с момента запуска VACUUM). Если по достижении autovacuum_freeze_max_age для таблицы ни разу не будет выполнена операция relfrozenxid, в скором времени для неё будет принудительно запущена автоочистка.

Если по какой-либо причине автоочистка не может вычистить старые значения XID из таблицы, система начинает выдавать предупреждающие сообщения, подобные приведённому ниже, когда самое старое значение XID в базе данных оказывается в 11 миллионах транзакций от точки зацикливания:

```
ПРЕДУПРЕЖДЕНИЕ:  база данных "mydb" должна быть очищена (предельное число транзакций:
 10985967)
ПОДСКАЗКА:  Во избежание отключения базы данных выполните очистку (VACUUM) всей базы.
```

(Проблему можно решить, как предлагает подсказка, запустив VACUUM вручную; однако учтите, что выполнять VACUUM должен суперпользователь, в противном случае эта процедура не сможет обработать системные каталоги и, следовательно, не сможет увеличить значение datfrozenxid для базы данных.) Если эти предупреждения игнорировать, система отключится и не будет начинать никаких транзакций, как только до точки зацикливания останется менее 1 миллиона транзакций:

```
ОШИБКА:  база данных не принимает команды во избежание потери данных из-за зацикливания
 в БД "mydb"
ПОДСКАЗКА:  Остановите управляющий процесс (postmaster) и выполните очистку (VACUUM)
 базы данных в однопользовательском режиме.
```

Резерв в 1 миллион транзакций позволяет администратору провести восстановление без потери данных, выполнив необходимые команды VACUUM вручную. Однако поскольку после безопасной остановки система не будет исполнять команды, администратору останется только перезапустить

сервер в однопользовательском режиме, чтобы запустить VACUUM. За подробной информацией об использовании однопользовательского режима обратитесь к странице справки по postgres.

23.1.5.1. Мультитранзакции и зацикливание

Идентификаторы мультитранзакций используются для поддержки блокировки строк несколькими транзакциями одновременно. Поскольку в заголовке строки есть только ограниченное пространство для хранения информации о блокировках, в нём указывается «идентификатор множественной транзакции», или идентификатор мультитранзакции для краткости, когда строку блокируют одновременно несколько транзакций. Информация о том, какие именно идентификаторы транзакций относятся к определённой мультитранзакции, хранится отдельно в подкаталоге pg_multixact, а в поле xmax в заголовке строки сохраняется только идентификатор мультитранзакции. Как и идентификаторы транзакций, идентификаторы мультитранзакций исполнены в виде 32-разрядного счётчика и хранятся аналогично, что требует аккуратного управления их возрастом, очисткой хранилища и предотвращением зацикливаний. Существует отдельная область, в которой содержится список членов каждой мультитранзакции, где счётчики также 32-битные и требуют должного контроля.

Когда VACUUM сканирует какую-либо часть таблицы, каждый идентификатор мультитранзакции старее чем vacuum_multixact_freeze_min_age заменяется другим значением, которое может быть нулевым, идентификатором одиночной транзакции или новым идентификатором мультитранзакции. Для каждой таблицы в pg_class.relminmxid хранится самый старый возможный идентификатор мультитранзакции, всё ещё задействованный в какой-либо строке этой таблицы. Если это значение оказывается старее vacuum_multixact_freeze_table_age, выполняется агрессивная очистка. Как рассказывалось в предыдущем разделе, при агрессивной очистке будут пропускаться только те страницы, которые считаются полностью замороженными. Узнать возраст pg_class.relminmxid можно с помощью функции mxid_age().

Благодаря агрессивным операциям VACUUM, вне зависимости от их причины, это значение для таблицы будет увеличиваться. В конце концов, по мере сканирования всех таблиц во всех базах данных и увеличения их старейших значений мультитранзакций, информация о старых мультитранзакциях может быть удалена с диска.

В качестве меры защиты, агрессивное сканирование с целью очистки будет происходить для любой таблицы, возраст мультитранзакций которой больше, чем autovacuum_multixact_freeze_max_age. Агрессивное сканирование также будет выполняться постепенно со всеми таблицами, начиная с имеющих старейшие мультитранзакции, если объём занятой области членов мультитранзакций превышает 50% от объёма адресуемого пространства. Эти два варианта агрессивного сканирования осуществляются, даже если процесс автоочистки отключён.

23.1.6. Демон автоочистки

В Postgres Pro имеется не обязательная, но настоятельно рекомендуемая к использованию функция, называемая *автоочисткой*, предназначение которой — автоматизировать выполнение команд VACUUM и ANALYZE. Когда автоочистка включена, она проверяет, в каких таблицах было вставлено, изменено или удалено много строк. При этих проверках используются средства сбора статистики; поэтому автоочистка будет работать, только если параметр track_counts имеет значение true. В конфигурации по умолчанию автоочистка включена и соответствующие параметры имеют подходящие значения.

«Демон автоочистки» на самом деле состоит из нескольких процессов. Существует постоянный фоновый процесс, называемый *процессом запуска автоочистки*, который отвечает за запуск *рабочих процессов автоочистки* для всех баз данных. Этот контролирующий процесс распределяет работу по времени, стараясь запускать рабочий процесс для каждой базы данных каждые autovacuum_naptime секунд. (Следовательно, если всего имеется *N* баз данных, новый рабочий процесс будет запускаться каждые autovacuum_naptime/*N* секунд.) Одновременно могут выполняться до autovacuum_max_workers рабочих процессов. Если число баз данных, требующих обработки, превышает autovacuum_max_workers, обработка следующей базы начинается сразу по завершении первого рабочего процесса. Каждый рабочий процесс проверяет все таблицы в

своей базе данных и в случае необходимости выполняет VACUUM и/или ANALYZE. Для отслеживания действий рабочих процессов можно установить параметр log_autovacuum_min_duration.

Если в течение короткого промежутка времени потребность в очистке возникает для нескольких больших таблиц, все рабочие процессы автоочистки могут продолжительное время заниматься очисткой только этих таблиц. В результате другие таблицы и базы данных будут ожидать очистки, пока не появится свободный рабочий процесс. Число рабочих процессов для одной базы не ограничивается, при этом каждый процесс старается не повторять работу, только что выполненную другими. Заметьте, что в ограничениях max_connections или superuser_reserved_connections число выполняющихся рабочих процессов не учитывается.

Для таблиц с relfrozenxid, устаревшим более чем на autovacuum_freeze_max_age транзакций, очистка выполняется всегда (это также применимо к таблицам, для которых максимальный порог заморозки был изменён через параметры хранения; см. ниже). В противном случае очистка таблицы производится, если количество кортежей, устаревших с момента последнего выполнения VACUUM, превышает «пороговое значение очистки». Пороговое значение очистки определяется как:

```
порог очистки = базовый порог очистки + коэффициент доли для очистки * количество
 кортежей
```

где базовый порог очистки — значение autovacuum_vacuum_threshold, коэффициент доли — autovacuum_vacuum_scale_factor, а количество кортежей — pg_class.reltuples. Количество устаревших кортежей получается от сборщика статистики; оно представляет собой приблизительное число, обновляемое после каждой операции UPDATE и DELETE. (Точность не гарантируется, потому что при большой нагрузке часть информации может быть утеряна.) Если значение relfrozenxid для таблицы старее vacuum_freeze_table_age транзакций, производится агрессивная очистка с целью заморозить старые версии строк и увеличить значение relfrozenxid; в противном случае сканируются только страницы, изменённые после последней очистки.

Для выполнения сбора статистики используется аналогичное условие: пороговое значение, определяемое как:

```
порог анализа = базовый порог анализа + коэффициент доли для анализа * количество
 кортежей
```

сравнивается с общим количеством кортежей добавленных, изменённых или удалённых после последнего выполнения ANALYZE.

Процесс автоочистки не затрагивает секционированные таблицы. Сбор статистики должен выполняться процедурой ANALYZE, запускаемой вручную после первого наполнения таблицы, а также всякий раз после значительного изменения распределения данных по секциям.

Автоочистка не обрабатывает временные таблицы. Поэтому очистку и сбор статистики в них нужно производить с помощью SQL-команд в обычном сеансе.

Используемые по умолчанию пороговые значения и коэффициенты берутся из postgresql.conf, однако их (и многие другие параметры, управляющие автоочисткой) можно переопределить для каждой таблицы; за подробностями обратитесь к Подразделу «Параметры хранения». Если какие-либо значения определены через параметры хранения таблицы, при обработке этой таблицы действуют они, а в противном случае — глобальные параметры. За более подробной информацией о глобальных параметрах обратитесь к Разделу 18.10.

Когда выполняются несколько рабочих процессов, параметры задержки автоочистки по стоимости (см. Подраздел 18.4.4) «распределяются» между всеми этими процессами, так что общее воздействие на систему остаётся неизменным, независимо от их числа. Однако этот алгоритм распределения нагрузки не учитывает процессы, обрабатывающие таблицы с индивидуальными значениями параметров хранения autovacuum_vacuum_cost_delay и autovacuum_vacuum_cost_limit.

Рабочие процессы автоочистки обычно не мешают выполнению других команд. Если какой-либо процесс попытается получить блокировку, конфликтующую с блокировкой SHARE UPDATE

EXCLUSIVE, которая удерживается в ходе автоочистки, автоочистка прервётся и процесс получит нужную ему блокировку. Конфликтующие режимы блокировок отмечены в Таблице 13.2. Однако если автоочистка выполняется для предотвращения зацикливания идентификаторов транзакций (т. е. описание запроса автоочистки в представлении pg_stat_activity заканчивается на (to prevent wraparound)), автоочистка не прерывается без ручного вмешательства.

> **Предупреждение**
>
> При частом выполнении таких команд, как ANALYZE, которые затребуют блокировки, конфликтующие с SHARE UPDATE EXCLUSIVE, может получиться так, что автоочистка не будет успевать завершаться в принципе.

23.2. Регулярная переиндексация

В некоторых ситуациях стоит периодически перестраивать индексы, выполняя команду REINDEX или последовательность отдельных шагов по восстановлению индексов.

Страницы индексов на основе B-деревьев, которые стали абсолютно пустыми, могут быть использованы повторно. Однако возможность неэффективного использования пространства всё же остаётся: если со страницы были удалены почти все, но не все ключи индекса, страница всё равно остаётся занятой. Следовательно, шаблон использования, при котором со временем удаляются многие, но не все ключи в каждом диапазоне, приведёт к неэффективному расходованию пространства. В таких случаях рекомендуется периодически проводить переиндексацию.

Возможность потери пространства в индексах на основе не B-деревьев глубоко не исследовалась. Поэтому имеет смысл периодически отслеживать физический размер индекса, когда применяется индекс такого типа.

Кроме того, с B-деревьями доступ по недавно построенному индексу осуществляется немного быстрее, нежели доступ по индексу, который неоднократно изменялся, поскольку в недавно построенном индексе страницы, близкие логически, обычно расположены так же близко и физически. (Это соображение неприменимо к индексам, которые основаны не на B-деревьях.) Поэтому периодически проводить переиндексацию стоит хотя бы для того, чтобы увеличить скорость доступа.

Команду REINDEX проста и безопасна для использования в любых случаях. Но так как она требует исключительной блокировки таблицы, часто предпочтительнее перестраивать индекс в несколько этапов, включающих создание и замену индекса. Типы индексов, которые поддерживает CREATE INDEX с указанием CONCURRENTLY, можно построить именно так. Если это удаётся и получен рабочий индекс, изначальный индекс можно заменить им, выполнив ALTER INDEX и DROP INDEX. Когда индекс используется для обеспечения уникальности или других ограничений, может потребоваться команда ALTER TABLE, чтобы поменять существующее ограничение на то, что обеспечивает новый индекс. Обстоятельно продумайте эту многоходовую процедуру, прежде чем выполнять её, так как не все индексы можно перестроить таким образом, и предусмотрите обработку ошибок.

23.3. Обслуживание журнала

Журнал сервера базы данных желательно сохранять где-либо, а не просто сбрасывать его в /dev/null. Этот журнал бесценен при диагностике проблем.

> **Примечание**
>
> Журнал сервера может содержать конфиденциальную информацию и должен быть защищён, где бы он ни хранился и куда бы ни передавался. Например, операторы DDL могут содержать незашифрованные пароли или другие данные аутентификации. В операторах, выводимых на уровне ERROR, может отображаться исходный код SQL для приложений, а также могут

содержаться фрагменты строк данных. Так как журнал собственно предназначен для записи данных, событий и связанной с ними информации, это не является уязвимостью или дефектом. Вам следует позаботиться о том, чтобы к журналам сервера получали доступ только лица с соответствующими полномочиями.

Журнал сервера может быть очень объёмным (особенно при высоких уровнях отладки), так что хранить его неограниченно долго вы вряд ли захотите. Поэтому необходимо организовать *ротацию* журнальных файлов так, чтобы новые файлы создавались, а старые удалялись через разумный промежуток времени.

Если просто направить stderr команды `postgres` в файл, вы получите в нём журнал сообщений, но очистить этот файл можно будет, только если остановить и перезапустить сервер. Это может быть допустимо при использовании Postgres Pro в среде разработки, но вряд ли такой вариант будет приемлемым в производственной среде.

Лучшим подходом будет перенаправление вывода сервера stderr в какую-либо программу ротации журнальных файлов. Существует и встроенное средство ротации журнальных файлов, которое можно использовать, установив для параметра `logging_collector` значение `true` в `postgresql.conf`. Параметры, управляющие этой программой, описаны в Подразделе 18.8.1. Этот подход также можно использовать для получения содержимого журнала в формате CSV (значения, разделённые запятыми).

Вы также можете предпочесть внешнюю программу ротации журналов, если вы используете её для другого серверного ПО. Например, для Postgres Pro можно применить программу rotatelogs, включённую в дистрибутив Apache. Один из вариантов такого использования — направить поток сервера stderr на вход такой программе. Если вы запускаете сервер с помощью `pg_ctl`, поток stderr уже выводится в stdout, так что вам нужна простая команда перенаправления:

```
pg_ctl start | rotatelogs /var/log/pgsql_log 86400
```

Вы можете скомбинировать эти подходы, настроив программу logrotate так, чтобы она собирала файлы журналов, которые записывает встроенный в Postgres Pro сборщик сообщений. В этом случае имена и расположение файлов журналов определяет сборщик сообщений, а logrotate периодически архивирует эти файлы. Когда logrotate производит ротацию журналов, важно, чтобы приложение выводило дальнейшие сообщения в новый файл. Обычно это делает скрипт `postrotate`, передающий сигнал `SIGHUP` приложению, которое в свою очередь заново открывает файл журнала. В Postgres Pro вместо этого вы можете выполнить pg_ctl с указанием `logrotate`. Когда выполняется эта команда, сервер либо переключается на новый файл журнала, либо заново открывает существующий, в зависимости от конфигурации журналирования (см. Подраздел 18.8.1).

Примечание

Когда используются статические имена файлов журналов, сервер может столкнуться с ошибкой при открытии файла, если будет достигнуто ограничение на максимальное число открытых файлов или переполнится таблица файлов. В этом случае сообщения будут продолжать выводиться в старый журнал пока не произойдёт успешное переключение. Если программа logrotate сжимает файл журнала и затем удаляет его, сообщения сервера, выводимые в этом интервале времени, могут быть потеряны. Чтобы избежать этого, можно сделать так, чтобы сборщик сообщений выбирал динамические имена файлов, а скрипт `prerotate` игнорировал открытые файлы журналов.

Ещё одно решение промышленного уровня заключается в передаче журнала в syslog, чтобы ротацией файлов занималась уже служба syslog. Для этого присвойте параметру конфигурации `log_destination` значение `syslog` (для вывода журнала только в syslog) в `postgresql.conf`. Затем вы сможете посылать сигнал `SIGHUP` службе syslog, когда захотите принудительно начать запись нового журнального файла. Если вы хотите автоматизировать ротацию журнальных файлов,

программу logrotate можно настроить и для работы с журнальными файлами, которые формирует syslog.

Однако во многих системах, а особенно с большими сообщениями, syslog работает не очень надёжно; он может обрезать или терять сообщения как раз тогда, когда они вам нужны. Кроме того, в Linux, syslog> сбрасывает каждое сообщение на диск, от чего страдает производительность. (Для отключения этой синхронной записи можно добавить «–» перед именем файла в файле конфигурации syslog.)

Обратите внимание, что все описанные выше решения обеспечивают создание новых журнальных файлов через задаваемые промежутки времени, но не удаление старых, ставших бесполезными файлов журналов. Возможно, вы захотите создать задание для периодического удаления старых файлов. Кроме того, вы можете настроить программу ротации файлов так, чтобы старые файлы журналов циклически перезаписывались.

Также вам может быть полезен *pgBadger* — инструмент для сложного анализа файлов журнала. Кроме того, *check_postgres* может посылать уведомления в Nagios, когда в журнале появляются важные сообщения, а также при обнаружении других нестандартных ситуаций.

Глава 24. Резервное копирование и восстановление

Как и всё, что содержит важные данные, базы данных Postgres Pro следует регулярно сохранять в резервной копии. Хотя эта процедура по существу проста, важно чётко понимать лежащие в её основе приёмы и положения.

Существует три фундаментально разных подхода к резервному копированию данных в Postgres Pro:

- Выгрузка в SQL
- Копирование на уровне файлов
- Непрерывное архивирование

Каждый из них имеет свои сильные и слабые стороны; все они обсуждаются в следующих разделах.

24.1. Выгрузка в SQL

Идея, стоящая за этим методом, заключается в генерации текстового файла с командами SQL, которые при выполнении на сервере пересоздадут базу данных в том же самом состоянии, в котором она была на момент выгрузки. Postgres Pro предоставляет для этой цели вспомогательную программу pg_dump. Простейшее применение этой программы выглядит так:

```
pg_dump имя_базы > файл_дампа
```

Как видите, pg_dump записывает результаты своей работы в устройство стандартного вывода. Далее будет рассмотрено, чем это может быть полезно. В то время как вышеупомянутая команда создаёт текстовый файл, pg_dump может создать файлы и в других форматах, которые допускают параллельную обработку и более гибкое управление восстановлением объектов.

Программа pg_dump является для Postgres Pro обычным клиентским приложением (хотя и весьма умным). Это означает, что вы можете выполнять процедуру резервного копирования с любого удалённого компьютера, если имеете доступ к нужной базе данных. Но помните, что pg_dump не использует для своей работы какие-то специальные привилегии. В частности, ей обычно требуется доступ на чтение всех таблиц, которые вы хотите выгрузить, так что для копирования всей базы данных практически всегда её нужно запускать с правами суперпользователя СУБД. (Если у вас нет достаточных прав для резервного копирования всей базы данных, вы тем не менее можете сделать резервную копию той части базы, доступ к которой у вас есть, используя такие параметры, как `-n` *`схема`* или `-t` *`таблица`*.)

Указать, к какому серверу должна подключаться программа pg_dump, можно с помощью аргументов командной строки `-h` *`сервер`* и `-p` *`порт`*. По умолчанию в качестве сервера выбирается localhost или значение, указанное в переменной окружения `PGHOST`. Подобным образом, по умолчанию используется порт, заданный в переменной окружения `PGPORT`, а если она не задана, то порт, указанный по умолчанию при компиляции. (Для удобства при компиляции сервера обычно устанавливается то же значение по умолчанию.)

Как и любое другое клиентское приложение Postgres Pro, pg_dump по умолчанию будет подключаться к базе данных с именем пользователя, совпадающим с именем текущего пользователя операционной системы. Чтобы переопределить имя, либо добавьте параметр `-U`, либо установите переменную окружения `PGUSER`. Помните, что pg_dump подключается к серверу через обычные механизмы проверки подлинности клиента (которые описываются в Главе 19).

Важное преимущество pg_dump в сравнении с другими методами резервного копирования, описанными далее, состоит в том, что вывод pg_dump обычно можно загрузить в более новые версии Postgres Pro, в то время как резервная копия на уровне файловой системы и непрерывное архивирование жёстко зависят от версии сервера. Также, только метод с применением pg_dump будет работать при переносе базы данных на другую машинную архитектуру, например, при переносе с 32-битной на 64-битную версию сервера.

Дампы, создаваемые pg_dump, являются внутренне согласованными, то есть, дамп представляет собой снимок базы данных на момент начала запуска pg_dump. pg_dump не блокирует другие операции с базой данных во время своей работы. (Исключение составляют операции, которым нужна исключительная блокировка, как например, большинство форм команды `ALTER TABLE`.)

24.1.1. Восстановление дампа

Текстовые файлы, созданные pg_dump, предназначаются для последующего чтения программой psql. Общий вид команды для восстановления дампа:

```
psql имя_базы < файл_дампа
```

где *файл_дампа* — это файл, содержащий вывод команды pg_dump. База данных, заданная параметром *имя_базы*, не будет создана данной командой, так что вы должны создать её сами из базы `template0` перед запуском psql (например, с помощью команды `createdb -T template0 имя_базы`). Программа psql принимает параметры, указывающие сервер, к которому осуществляется подключение, и имя пользователя, подобно pg_dump. За дополнительными сведениями обратитесь к справке по psql. Дампы, выгруженные не в текстовом формате, восстанавливаются утилитой pg_restore.

Перед восстановлением SQL-дампа все пользователи, которые владели объектами или имели права на объекты в выгруженной базе данных, должны уже существовать. Если их нет, при восстановлении будут ошибки пересоздания объектов с изначальными владельцами и/или правами. (Иногда это желаемый результат, но обычно нет).

По умолчанию, если происходит ошибка SQL, программа psql продолжает выполнение. Если же запустить psql с установленной переменной `ON_ERROR_STOP`, это поведение поменяется и psql завершится с кодом 3 в случае возникновения ошибки SQL:

```
psql --set ON_ERROR_STOP=on имя_базы < файл_дампа
```

В любом случае вы получите только частично восстановленную базу данных. В качестве альтернативы можно указать, что весь дамп должен быть восстановлен в одной транзакции, так что восстановление либо полностью выполнится, либо полностью отменится. Включить данный режим можно, передав psql аргумент `-1` или `--single-transaction`. Выбирая этот режим, учтите, что даже незначительная ошибка может привести к откату восстановления, которое могло продолжаться несколько часов. Однако это всё же может быть предпочтительней, чем вручную вычищать сложную базу данных после частично восстановленного дампа.

Благодаря способности pg_dump и psql писать и читать каналы ввода/вывода, можно скопировать базу данных непосредственно с одного сервера на другой, например:

```
pg_dump -h host1 имя_базы | psql -h host2 имя_базы
```

> **Важно**
>
> Дампы, которые выдаёт pg_dump, содержат определения относительно `template0`. Это означает, что любые языки, процедуры и т. п., добавленные в базу через `template1`, pg_dump также выгрузит в дамп. Как следствие, если при восстановлении вы используете модифицированный `template1`, вы должны создать пустую базу данных из `template0`, как показано в примере выше.

После восстановления резервной копии имеет смысл запустить ANALYZE для каждой базы данных, чтобы оптимизатор запросов получил полезную статистику; за подробностями обратитесь к Подразделу 23.1.3 и Подразделу 23.1.6. Другие советы по эффективной загрузке больших объёмов данных в Postgres Pro вы можете найти в Разделе 14.4.

24.1.2. Использование pg_dumpall

Программа pg_dump выгружает только одну базу данных в один момент времени и не включает в дамп информацию о ролях и табличных пространствах (так как это информация уровня

кластера, а не самой базы данных). Для удобства создания дампа всего содержимого кластера баз данных предоставляется программа pg_dumpall, которая делает резервную копию всех баз данных кластера, а также сохраняет данные уровня кластера, такие как роли и определения табличных пространств. Простое использование этой команды:

```
pg_dumpall > файл_дампа
```

Полученную копию можно восстановить с помощью psql:

```
psql -f файл_дампа postgres
```

(В принципе, здесь в качестве начальной базы данных можно указать имя любой существующей базы, но если вы загружаете дамп в пустой кластер, обычно нужно использовать `postgres`). Восстанавливать дамп, который выдала pg_dumpall, всегда необходимо с правами суперпользователя, так как они требуются для восстановления информации о ролях и табличных пространствах. Если вы используете табличные пространства, убедитесь, что пути к табличным пространствам в дампе соответствуют новой среде.

pg_dumpall выдаёт команды, которые заново создают роли, табличные пространства и пустые базы данных, а затем вызывает для каждой базы pg_dump. Таким образом, хотя каждая база данных будет внутренне согласованной, состояние разных баз не будет синхронным.

Только глобальные данные кластера можно выгрузить, передав pg_dumpall ключ `--globals-only`. Это необходимо, чтобы полностью скопировать кластер, когда pg_dump выполняется для отдельных баз данных.

24.1.3. Управление большими базами данных

Некоторые операционные системы накладывают ограничение на максимальный размер файла, что приводит к проблемам при создании больших файлов с помощью pg_dump. К счастью, pg_dump может писать в стандартный вывод, так что вы можете использовать стандартные инструменты Unix для того, чтобы избежать потенциальных проблем. Вот несколько возможных методов:

Используйте сжатые дампы. Вы можете использовать предпочитаемую программу сжатия, например gzip:

```
pg_dump имя_базы | gzip > имя_файла.gz
```

Затем загрузить сжатый дамп можно командой:

```
gunzip -c имя_файла.gz | psql имя_базы
```

или:

```
cat имя_файла.gz | gunzip | psql имя_базы
```

Используйте `split`. Команда `split` может разбивать выводимые данные на небольшие файлы, размер которых удовлетворяет ограничению нижележащей файловой системы. Например, чтобы получить части по 2 гигабайта:

```
pg_dump имя_базы | split -b 2G - имя_файла
```

Восстановить их можно так:

```
cat имя_файла* | psql имя_базы
```

Использовать GNU split можно вместе с gzip:

```
pg_dump имя_базы | split -b 2G --filter='gzip > $FILE.gz'
```

Восстановить данные после такого разбиения можно с помощью команды `zcat`.

Используйте специальный формат дампа pg_dump. Если при сборке Postgres Pro была подключена библиотека zlib, дамп в специальном формате будет записываться в файл в сжатом виде. В таком формате размер файла дампа будет близок к размеру, полученному с применением

gzip, но он лучше тем, что позволяет восстанавливать таблицы выборочно. Следующая команда выгружает базу данных в специальном формате:

```
pg_dump -Fc имя_базы > имя_файла
```

Дамп в специальном формате не является скриптом для psql и должен восстанавливаться с помощью команды pg_restore, например:

```
pg_restore -d имя_базы имя_файла
```

За подробностями обратитесь к справке по командам pg_dump и pg_restore.

Для очень больших баз данных может понадобиться сочетать split с одним из двух других методов.

Используйте возможность параллельной выгрузки в pg_dump. Чтобы ускорить выгрузку большой БД, вы можете использовать режим параллельной выгрузки в pg_dump. При этом одновременно будут выгружаться несколько таблиц. Управлять числом параллельных заданий позволяет параметр -j. Параллельная выгрузка поддерживается только для формата архива в каталоге.

```
pg_dump -j число -F d -f выходной_каталог имя_базы
```

Вы также можете восстановить копию в параллельном режиме с помощью pg_restore -j. Это поддерживается для любого архива в формате каталога или специальном формате, даже если архив создавался не командой pg_dump -j.

24.2. Резервное копирование на уровне файлов

Альтернативной стратегией резервного копирования является непосредственное копирование файлов, в которых Postgres Pro хранит содержимое базы данных; в Разделе 17.2 рассказывается, где находятся эти файлы. Вы можете использовать любой способ копирования файлов по желанию, например:

```
tar -cf backup.tar /usr/local/pgsql/data
```

Однако существуют два ограничения, которые делают этот метод непрактичным или как минимум менее предпочтительным по сравнению с pg_dump:

1. Чтобы полученная резервная копия была годной, сервер баз данных *должен* быть остановлен. Такие полумеры, как запрещение всех подключений к серверу, работать *не* будут (отчасти потому что tar и подобные средства не получают мгновенный снимок состояния файловой системы, но ещё и потому, что в сервере есть внутренние буферы). Узнать о том, как остановить сервер, можно в Разделе 17.5. Необходимо отметить, что сервер нужно будет остановить и перед восстановлением данных.
2. Если вы ознакомились с внутренней организацией базы данных в файловой системе, у вас может возникнуть соблазн скопировать или восстановить только отдельные таблицы или базы данных в соответствующих файлах или каталогах. Это *не* будет работать, потому что информацию, содержащуюся в этих файлах, нельзя использовать без файлов журналов транзакций, pg_xact/*, которые содержат состояние всех транзакций. Без этих данных файлы таблиц непригодны к использованию. Разумеется также невозможно восстановить только одну таблицу и соответствующие данные pg_xact, потому что в результате нерабочими станут все другие таблицы в кластере баз данных. Таким образом, копирование на уровне файловой системы будет работать, только если выполняется полное копирование и восстановление всего кластера баз данных.

Ещё один подход к резервному копированию файловой системы заключается в создании «целостного снимка» каталога с данными, если это поддерживает файловая система (и вы склонны считать, что эта функциональность реализована корректно). Типичная процедура включает создание «замороженного снимка» тома, содержащего базу данных, затем копирование всего каталога с данными (а не его избранных частей, см. выше) из этого снимка на устройство резервного копирования, и наконец освобождение замороженного снимка. При этом сервер базы данных может не прекращать свою работу. Однако резервная копия, созданная таким способом, содержит файлы базы данных в таком состоянии, как если бы сервер баз данных не был остановлен

штатным образом; таким образом, когда вы запустите сервер баз данных с сохранёнными данными, он будет считать, что до этого процесс сервера был прерван аварийно, и будет накатывать журнал WAL. Это не проблема, просто имейте это в виду (и обязательно включите файлы WAL в резервную копию). Чтобы сократить время восстановления, можно выполнить команду `CHECKPOINT` перед созданием снимка.

Если ваша база данных размещена в нескольких файловых системах, получить в точности одновременно замороженные снимки всех томов может быть невозможно. Например, если файлы данных и журналы WAL находятся на разных дисках или табличные пространства расположены в разных файловых системах, резервное копирование со снимками может быть неприменимо, потому что снимки *должны* быть одновременными. В таких ситуациях очень внимательно изучите документацию по вашей файловой системе, прежде чем довериться технологии согласованных снимков.

Если одновременные снимки невозможны, остаётся вариант с остановкой сервера баз данных на время, достаточное для получения всех замороженных снимков. Другое возможное решение — получить базовую копию путём непрерывного архивирования (см. Подраздел 24.3.2), такие резервные копии не могут пострадать от изменений файловой системы в процессе резервного копирования. Для этого требуется включить непрерывное архивирование только на время резервного копирования; для восстановления применяется процедура восстановления из непрерывного архива (Подраздел 24.3.4).

Ещё один вариант — копировать содержимое файловой системы с помощью rsync. Для этого rsync запускается сначала во время работы сервера баз данных, а затем сервер останавливается на время, достаточное для запуска `rsync --checksum`. (Ключ `--checksum` необходим, потому что `rsync` различает время только с точностью до секунд.) Во второй раз rsync отработает быстрее, чем в первый, потому что скопировать надо будет относительно немного данных; и в итоге будет получен согласованный результат, так как сервер был остановлен. Данный метод позволяет получить копию на уровне файловой системы с минимальным временем простоя.

Обратите внимание, что размер копии на уровне файлов обычно больше, чем дампа SQL. (Программе pg_dump не нужно, например, записывать содержимое индексов, достаточно команд для их пересоздания). Однако копирование на уровне файлов может выполняться быстрее.

24.3. Непрерывное архивирование и восстановление на момент времени (Point-in-Time Recovery, PITR)

Всё время в процессе работы Postgres Pro ведёт *журнал предзаписи* (WAL), который расположен в подкаталоге `pg_wal/` каталога с данными кластера баз данных. В этот журнал записываются все изменения, вносимые в файлы данных. Прежде всего, журнал существует для безопасного восстановления после краха сервера: если происходит крах, целостность СУБД может быть восстановлена в результате «воспроизведения» записей, зафиксированных после последней контрольной точки. Однако наличие журнала делает возможным использование третьей стратегии копирования баз данных: можно сочетать резервное копирование на уровне файловой системы с копированием файлов WAL. Если потребуется восстановить данные, мы можем восстановить копию файлов, а затем воспроизвести журнал из скопированных файлов WAL, и таким образом привести систему в нужное состояние. Такой подход более сложен для администрирования, чем любой из описанных выше, но он имеет значительные преимущества:

- В качестве начальной точки для восстановления необязательно иметь полностью согласованную копию на уровне файлов. Внутренняя несогласованность копии будет исправлена при воспроизведении журнала (практически то же самое происходит при восстановлении после краха). Таким образом, согласованный снимок файловой системы не требуется, вполне можно использовать tar или похожие средства архивации.
- Поскольку при воспроизведении можно обрабатывать неограниченную последовательность файлов WAL, непрерывную резервную копию можно получить, просто продолжая архивировать файлы WAL. Это особенно ценно для больших баз данных, полные резервные копии которых делать как минимум неудобно.

- Воспроизводить все записи WAL до самого конца нет необходимости. Воспроизведение можно остановить в любой точке и получить целостный снимок базы данных на этот момент времени. Таким образом, данная технология поддерживает *восстановление на момент времени*: можно восстановить состояние базы данных на любое время с момента создания резервной копии.
- Если непрерывно передавать последовательность файлов WAL другому серверу, получившему данные из базовой копии того же кластера, получается система *тёплого резерва*: в любой момент мы можем запустить второй сервер и он будет иметь практически текущую копию баз данных.

> **Примечание**
>
> Программы pg_dump и pg_dumpall не создают копии на уровне файловой системы и не могут применяться как часть решения по непрерывной архивации. Создаваемые ими копии являются логическими и не содержат информации, необходимой для воспроизведения WAL.

Как и обычное резервное копирование файловой системы, этот метод позволяет восстанавливать только весь кластер баз данных целиком, но не его части. Кроме того, для архивов требуется большое хранилище: базовая резервная копия может быть объёмной, а нагруженные системы будут генерировать многие мегабайты трафика WAL, который необходимо архивировать. Тем не менее этот метод резервного копирования предпочитается во многих ситуациях, где необходима высокая надёжность.

Для успешного восстановления с применением непрерывного архивирования (также называемого «оперативным резервным копированием» многими разработчиками СУБД), вам необходима непрерывная последовательность заархивированных файлов WAL, начинающаяся не позже, чем с момента начала копирования. Так что для начала вы должны настроить и протестировать процедуру архивирования файлов WAL *до того*, как получите первую базовую копию. Соответственно, сначала мы обсудим механику архивирования файлов WAL.

24.3.1. Настройка архивирования WAL

В абстрактном смысле, запущенная СУБД Postgres Pro производит неограниченно длинную последовательность записей WAL. СУБД физически делит эту последовательность на *файлы-сегменты* WAL, которые обычно имеют размер в 16 МиБ (хотя размер сегмента может быть изменён при сборке Postgres Pro). Файлы-сегменты получают цифровые имена, которые отражают их позицию в абстрактной последовательности WAL. Когда архивирование WAL не применяется, система обычно создаёт только несколько файлов-сегментов и затем «перерабатывает» их, меняя номер в имени более не нужного файла-сегмента на больший. Предполагается, что файлы-сегменты, чьё содержимое предшествует последней контрольной точке, уже не представляют интереса и могут быть переработаны.

При архивировании данных WAL необходимо считывать содержимое каждого файла-сегмента, как только он заполняется, и сохранять эти данные куда-то, прежде чем файл-сегмент будет переработан и использован повторно. В зависимости от применения и доступного аппаратного обеспечения, возможны разные способы «сохранить данные куда-то»: можно скопировать файлы-сегменты в смонтированный по NFS каталог на другую машину, записать их на ленту (убедившись, что у вас есть способ идентифицировать исходное имя каждого файла) или собрать их в пакет и записать на CD, либо какие-то совсем другие варианты. Чтобы у администратора баз данных была гибкость в этом плане, Postgres Pro пытается не делать каких-либо предположений о том, как будет выполняться архивация. Вместо этого, Postgres Pro позволяет администратору указать команду оболочки, которая будет запускаться для копирования завершённого файла-сегмента в нужное место. Эта команда может быть простой как `cp`, а может вызывать сложный скрипт оболочки — это решать вам.

Чтобы включить архивирование WAL, установите в параметре конфигурации wal_level уровень `replica` (или выше), в archive_mode — значение `on`, и задайте желаемую команду

оболочки в параметре archive_command. На практике эти параметры всегда задаются в файле postgresql.conf. В archive_command символы %p заменяются полным путём к файлу, подлежащему архивации, а %f заменяются только именем файла. (Путь задаётся относительно текущего рабочего каталога, т. е. каталога данных кластера). Если в команду нужно включить сам символ %, запишите %%. Простейшая команда, которая может быть полезна:

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/
%f'  # Unix
archive_command = 'copy "%p" "C:\\server\\archivedir\\%f"'  # Windows
```

Она будет копировать архивируемые сегменты WAL в каталог /mnt/server/archivedir. (Команда дана как пример, а не как рекомендация, и может работать не на всех платформах.) После замены параметров %p и %f фактически запускаемая команда может выглядеть так:

```
test ! -f /mnt/server/archivedir/00000001000000A900000065 && cp
 pg_wal/00000001000000A900000065 /mnt/server/archivedir/00000001000000A900000065
```

Подобная команда будет генерироваться для каждого следующего архивируемого файла.

Команда архивирования будет запущена от имени того же пользователя, от имени которого работает сервер Postgres Pro. Поскольку архивируемые последовательности файлов WAL фактически содержат всё, что есть в вашей базе данных, вам нужно будет защитить архивируемые данные от посторонних глаз; например, сохраните архив в каталог, чтение которого запрещено для группы и остальных пользователей.

Важно, чтобы команда архивирования возвращала нулевой код завершения, если и только если она завершилась успешно. Получив нулевой результат, Postgres Pro будет полагать, что файл успешно заархивирован и удалит его или переработает. Однако ненулевой код состояния скажет Postgres Pro, что файл не заархивирован; попытки заархивировать его будут периодически повторяться, пока это не удастся.

Команда архивирования обычно разрабатывается так, чтобы не допускать перезаписи любых существующих архивных файлов. Это важная мера безопасности, позволяющая сохранить целостность архива в случае ошибки администратора (например, если архивируемые данные двух разных серверов будут сохраняться в одном каталоге).

Рекомендуется протестировать команду архивирования, чтобы убедиться, что она действительно не перезаписывает существующие файлы и что она *возвращает ненулевое состояние в этом случае*. В показанной выше команде для Unix для этого добавлен отдельный шаг test. На некоторых платформах Unix у cp есть ключ -i, который позволяет сделать то же, но менее явно; но не проверив, какой код состояния при этом возвращается, полагаться на этот ключ не следует. (В частности, GNU cp возвратит нулевой код состояния, если используется ключ -i и целевой файл существует, а это *не то*, что нужно.)

Разрабатывая схему архивирования, подумайте, что произойдёт, если команда архивирования начнёт постоянно выдавать ошибку, потому что требуется вмешательство оператора или для архивирования не хватает места. Например, это может произойти, если вы записываете архивы на ленточное устройство без механизма автозамены; когда лента заполняется полностью, больше ничего архивироваться не будет, пока вы не замените кассету. Вы должны убедиться, что любые возникающие ошибки или обращения к человеку (оператору) обрабатываются так, чтобы проблема решалась достаточно быстро. Пока она не разрешится, каталог pg_wal/ продолжит наполняться файлами-сегментами WAL. (Если файловая система, в которой находится каталог pg_wal/ заполнится до конца, Postgres Pro завершит свою работу аварийно. Зафиксированные транзакции не потеряются, но база данных не будет работать, пока вы не освободите место.)

Не важно, с какой скоростью работает команда архивирования, если только она не ниже средней скорости, с которой сервер генерирует записи WAL. Обычно работа продолжается, даже если процесс архивирования немного отстаёт. Если же архивирование отстаёт значительно, это приводит к увеличению объёма данных, которые могут быть потеряны в случае аварии. При этом каталог pg_wal/ будет содержать большое количество ещё не заархивированных файлов-сегментов, которые в конце концов могут занять всё доступное дисковое пространство. Поэтому

рекомендуется контролировать процесс архивации и следить за тем, чтобы он выполнялся как задумано.

При написании команды архивирования вы должны иметь в виду, что имена файлов для архивирования могут иметь длину до 64 символов и содержать любые комбинации из цифр, точек и букв ASCII. Сохранять исходный относительный путь (`%p`) необязательно, но необходимо сохранять имя файла (`%f`).

Обратите внимание, что хотя архивирование WAL позволяет сохранить любые изменения данных, произведённые в базе данных Postgres Pro, оно не затрагивает изменения, внесённые в конфигурационные файлы (такие как `postgresql.conf`, `pg_hba.conf` и `pg_ident.conf`), поскольку эти изменения выполняются вручную, а не через SQL. Поэтому имеет смысл разместить конфигурационные файлы там, где они будут заархивированы обычными процедурами копирования файлов. Как перемещать конфигурационные файлы, рассказывается в Разделе 18.2.

Команда архивирования вызывается, только когда сегмент WAL заполнен до конца. Таким образом, если сервер постоянно генерирует небольшой трафик WAL (или есть продолжительные периоды, когда это происходит), между завершением транзакций и их безопасным сохранением в архиве может образоваться большая задержка. Чтобы ограничить время жизни неархивированных данных, можно установить archive_timeout, чтобы сервер переключался на новый файл сегмента WAL как минимум с заданной частотой. Заметьте, что неполные файлы, архивируемые досрочно из-за принудительного переключения по тайм-ауту, будут иметь тот же размер, что и заполненные файлы. Таким образом, устанавливать очень маленький `archive_timeout` неразумно — это приведёт к неэффективному заполнению архива. Обычно подходящее значение `archive_timeout` — минута или около того.

Также вы можете принудительно переключить сегмент WAL вручную с помощью `pg_switch_wal`, если хотите, чтобы только что завершённая транзакция заархивировалась как можно скорее. Другие полезные функции, относящиеся к управлению WAL, перечисляются в Таблице 9.79.

Когда `wal_level` имеет значение `minimal`, некоторые команды SQL выполняются в обход журнала WAL, как описывается в Подразделе 14.4.7. Если архивирование или потоковая репликация были включены во время выполнения таких операторов, WAL не будет содержать информацию, необходимую для восстановления. (На восстановление после краха это не распространяется). Поэтому `wal_level` можно изменить только при запуске сервера. Однако для изменения команды `archive_command` достаточно перезагрузить файл конфигурации. Если вы хотите на время остановить архивирование, это можно сделать, например, задав в качестве значения `archive_command` пустую строку (''). В результате файлы WAL будут накапливаться в каталоге `pg_wal/`, пока не будет восстановлена действующая команда `archive_command`.

24.3.2. Создание базовой резервной копии

Проще всего получить базовую резервную копию, используя программу pg_basebackup. Эта программа сохраняет базовую копию в виде обычных файлов или в архиве tar. Если гибкости pg_basebackup не хватает, вы также можете получить базовую резервную копию, используя низкоуровневый API (см. Подраздел 24.3.3).

Продолжительность создания резервной копии обычно не имеет большого значения. Однако если вы эксплуатируете сервер с отключённым режимом `full_page_writes`, вы можете заметить падение производительности в процессе резервного копирования, так как режим `full_page_writes` включается принудительно на время резервного копирования.

Чтобы резервной копией можно было пользоваться, нужно сохранить все файлы сегментов WAL, сгенерированные во время и после копирования файлов. Для облегчения этой задачи, процесс создания базовой резервной копии записывает *файл истории резервного копирования*, который немедленно сохраняется в области архивации WAL. Данный файл получает имя по имени файла первого сегмента WAL, который потребуется для восстановления скопированных файлов. Например, если начальный файл WAL назывался `000000010000123400000055CD`, файл истории резервного копирования получит имя `000000010000123400000055CD.007C9330.backup`. (Вторая часть имени файла обозначает точную позицию внутри файла WAL и обычно может быть

проигнорирована.) Как только вы заархивировали копии файлов данных и файлов сегментов WAL, полученных в процессе копирования (по сведениям в файле истории резервного копирования), все заархивированные сегменты WAL с именами, меньшими по номеру, становятся ненужными для восстановления файловой копии и могут быть удалены. Но всё же рассмотрите возможность хранения нескольких наборов резервных копий, чтобы быть абсолютно уверенными, что вы сможете восстановить ваши данные.

Файл истории резервного копирования — это просто небольшой текстовый файл. В него записывается метка, которая была передана pg_basebackup, а также время и текущие сегменты WAL в момент начала и завершения резервной копии. Если вы связали с данной меткой соответствующий файл дампа, то заархивированного файла истории достаточно, чтобы найти файл дампа, нужный для восстановления.

Поскольку необходимо хранить все заархивированные файлы WAL с момента последней базовой резервной копии, интервал базового резервного копирования обычно выбирается в зависимости от того, сколько места может быть выделено для архива файлов WAL. Также стоит отталкиваться от того, сколько вы готовы ожидать восстановления, если оно понадобится — системе придётся воспроизвести все эти сегменты WAL, а этот процесс может быть долгим, если с момента последней базовой копии прошло много времени.

24.3.3. Создание базовой резервной копии через низкоуровневый API

Процедура создания базовой резервной копии с использованием низкоуровневого API содержит чуть больше шагов, чем метод pg_basebackup, но всё же относительно проста. Очень важно, чтобы эти шаги выполнялись по порядку, и следующий шаг выполнялся, только если предыдущий успешен.

Резервное копирование на низком уровне можно произвести в монопольном или немонопольном режиме. Рекомендуется применять немонопольный метод, а монопольный считается устаревшим и в конце концов будет ликвидирован.

24.3.3.1. Немонопольное резервное копирование на низком уровне

Немонопольное резервное копирование позволяет параллельно запускать другие процессы копирования (используя тот же API или pg_basebackup).

1. Убедитесь, что архивирование WAL включено и работает.
2. Подключитесь к серверу (к любой базе данных) как пользователь с правами на выполнение pg_start_backup (суперпользователь или пользователь, которому дано право EXECUTE для этой функции) и выполните команду:

   ```
   SELECT pg_start_backup('label', false, false);
   ```

   где `label` — любая метка, по которой можно однозначно идентифицировать данную операцию резервного копирования. Соединение, через которое вызывается `pg_start_backup`, должно поддерживаться до окончания резервного копирования, иначе этот процесс будет автоматически прерван.

   По умолчанию `pg_start_backup` может выполняться длительное время. Это объясняется тем, что функция выполняет контрольную точку, а операции ввода/вывода, требуемые для этого, распределяются в интервале времени, по умолчанию равном половине интервала между контрольными точками (см. параметр checkpoint_completion_target). Обычно это вполне приемлемо, так как при этом минимизируется влияние на выполнение других запросов. Если же вы хотите начать резервное копирование максимально быстро, передайте во втором параметре `true`. В этом случае контрольная точка будет выполнена немедленно без ограничения объёма ввода/вывода.

   Третий параметр, имеющий значение `false`, указывает `pg_start_backup` начать немонопольное базовое копирование.

3. Скопируйте файлы, используя любое удобное средство резервного копирования, например, tar или cpio (не pg_dump или pg_dumpall). В процессе копирования останавливать работу базы данных не требуется, это ничего не даёт. В Подразделе 24.3.3.3 описано, что следует учитывать в процессе копирования.

4. Через то же подключение, что и раньше, выполните команду:

   ```
   SELECT * FROM pg_stop_backup(false, true);
   ```

   При этом сервер выйдет из режима резервного копирования. Ведущий сервер вместе с этим автоматически переключится на следующий сегмент WAL. На ведомом автоматическое переключение сегментов WAL невозможно, поэтому вы можете выполнить `pg_switch_wal` на ведущем, чтобы произвести переключение вручную. Такое переключение позволяет получить готовый к архивированию последний сегмент WAL, записанный в процессе резервного копирования.

   Функция `pg_stop_backup` возвратит одну строку с тремя значениями. Второе из них нужно записать в файл `backup_label` в корневой каталог резервной копии. Третье значение, если оно не пустое, должно быть записано в файл `tablespace_map`. Эти файлы крайне важны для восстановления копии и должны записываться байт за байтом без изменений, для чего может потребоваться открыть файл в двоичном редакторе.

5. После этого останется заархивировать файлы сегментов WAL, активных во время создания резервной копии, и процедура резервного копирования будет завершена. Функция `pg_stop_backup` в первом значении результата указывает, какой последний сегмент требуется для формирования полного набора файлов резервной копии. На ведущем сервере, если включён режим архивации (параметр `archive_mode`) и аргумент `wait_for_archive` равен `true`, функция `pg_stop_backup` не завершится, пока не будет заархивирован последний сегмент. На ведомом значением `archive_mode` должно быть `always`, чтобы `pg_stop_backup` ожидала архивации. Эти файлы будут заархивированы автоматически, поскольку также должна быть настроена команда `archive_command`. Чаще всего это происходит быстро, но мы советуем наблюдать за системой архивации и проверять, не возникают ли задержки. Если архивирование остановится из-за ошибок команды архивации, попытки архивации будут продолжаться до успешного завершения, и только тогда резервное копирование окончится. Если вы хотите ограничить время выполнения `pg_stop_backup`, установите соответствующее значение в `statement_timeout`, но заметьте, что в случае прерывания `pg_stop_backup` по времени резервная копия может оказаться негодной.

   Если в процедуре резервного копирования предусмотрено отслеживание и архивация всех файлов сегментов WAL, необходимых для резервной копии, то в аргументе `wait_for_archive` (по умолчанию равном true) можно передать false, чтобы функция `pg_stop_backup` завершилась сразу, как только в WAL будет помещена запись о завершении копирования. По умолчанию `pg_stop_backup` будет ждать окончания архивации всех файлов WAL, что может занять некоторое время. Использовать этот параметр следует с осторожностью: если архивация WAL не контролируется, в резервной копии могут оказаться не все необходимые файлы WAL и её нельзя будет восстановить.

24.3.3.2. Монопольное резервное копирование на низком уровне

Монопольное резервное копирование во многом похоже на немонопольное, но имеет несколько важных отличий. Такое копирование можно произвести только на ведущем сервере, и оно исключает одновременное выполнение других процессов копирования. До PostgreSQL 9.6 это был единственный возможный метод низкоуровневого копирования, но сейчас пользователям рекомендуется по возможности подкорректировать свои скрипты и перейти к использованию немонопольного варианта.

1. Убедитесь, что архивирование WAL включено и работает.

2. Подключитесь к серверу (к любой базе данных) как пользователь с правами на выполнение pg_start_backup (суперпользователь или пользователь, которому дано право EXECUTE для этой функции) и выполните команду:

   ```
   SELECT pg_start_backup('label');
   ```

где label — любая метка, по которой можно однозначно идентифицировать данную операцию резервного копирования. Функция pg_start_backup создаёт в каталоге кластера файл *метки резервного копирования*, называемый backup_label, в который помещается информация о резервной копии, включающая время начала и строку метки. Эта функция также создаёт в каталоге кластера файл *карты табличных пространств*, называемый tablespace_map, с информацией о символических ссылках табличных пространств в pg_tblspc/, если такие ссылки есть. Оба файла важны для целостности резервных копии и понадобятся при восстановлении.

По умолчанию pg_start_backup может выполняться длительное время. Это объясняется тем, что функция выполняет контрольную точку, а операции ввода/вывода, требуемые для этого, распределяются в интервале времени, по умолчанию равном половине интервала между контрольными точками (см. параметр checkpoint_completion_target). Обычно это вполне приемлемо, так как при этом минимизируется влияние на выполнение других запросов. Если же вы хотите начать резервное копирование максимально быстро, выполните:

```
SELECT pg_start_backup('label', true);
```

При этом контрольная точка будет выполнена как можно скорее.

3. Скопируйте файлы, используя любое удобное средство резервного копирования, например, tar или cpio (не pg_dump или pg_dumpall). В процессе копирования останавливать работу базы данных не требуется, это ничего не даёт. В Подразделе 24.3.3.3 описано, что следует учитывать в процессе копирования.

   Заметьте, что в случае сбоя сервера во время резервного копирования для успешного его перезапуска может потребоваться вручную удалить файл backup_label из каталога PGDATA.

4. Снова подключитесь к базе данных как пользователь с правами на выполнение pg_stop_backup (суперпользователь или пользователь, которому дано право EXECUTE для этой функции) и выполните команду:

   ```
   SELECT pg_stop_backup();
   ```

   Эта функция завершит режим резервного копирования и автоматически переключится на следующий сегмент WAL. Это переключение выполняется для того, чтобы файл последнего сегмента WAL, записанного во время копирования, был готов к архивации.

5. После этого останется заархивировать файлы сегментов WAL, активных во время создания резервной копии, и процедура резервного копирования будет завершена. Функция pg_stop_backup возвращает указание на файл последнего сегмента, который требуется для формирования полного набора файлов резервной копии. Если включён режим архивации (параметр archive_mode), функция pg_stop_backup не завершится, пока не будет заархивирован последний сегмент. В этом случае файлы будут заархивированы автоматически, поскольку также должна быть настроена команда archive_command. Чаще всего это происходит быстро, но мы советуем наблюдать за системой архивации и проверять, не возникают ли задержки. Если архивирование остановится из-за ошибок команды архивации, попытки архивации будут продолжаться до успешного завершения, и только тогда резервное копирование окончится. Если вы хотите ограничить время выполнения pg_stop_backup, установите соответствующее значение в statement_timeout, но заметьте, что в случае прерывания pg_stop_backup по времени резервная копия может оказаться негодной.

24.3.3.3. Копирование каталога данных

Некоторые средства резервного копирования файлов выдают предупреждения или ошибки, если файлы, которые они пытаются скопировать, изменяются в процессе копирования. При получении базовой резервной копии активной базы данных это вполне нормально и не является ошибкой. Однако вам нужно знать, как отличить ошибки такого рода от реальных ошибок. Например, некоторые версии rsync возвращают отдельный код завершения для ситуации «исчезнувшие исходные файлы», и вы можете написать управляющий скрипт, который примет этот код как не ошибочный. Также некоторые версии GNU tar возвращают код завершения, неотличимый от кода фатальной ошибки, если файл был усечён, когда tar копировал его. К счастью, GNU tar версий 1.16 и более поздних завершается с кодом 1, если файл был изменён во время копирования, и 2 в случае

других ошибок. С GNU tar версии 1.23 и более поздними, вы можете использовать следующие ключи `--warning=no-file-changed --warning=no-file-removed`, чтобы скрыть соответствующие предупреждения.

Убедитесь, что ваша резервная копия включает все файлы из каталога кластера баз данных (например, `/usr/local/pgsql/data`). Если вы используете табличные пространства, которые находятся не внутри этого каталога, не забудьте включить и их в резервную копию (также важно, чтобы при создании резервной копии символьные ссылки сохранялись как ссылки, иначе табличные пространства будут повреждены при восстановлении).

Однако следует исключить из резервной копии файлы в подкаталоге данных кластера `pg_wal/`. Эту небольшую корректировку стоит внести для снижения риска ошибок при восстановлении. Это легко организовать, если `pg_wal/` — символическая ссылка на каталог за пределами каталога данных (так часто делают из соображений производительности). Также имеет смысл исключить файлы `postmaster.pid` и `postmaster.opts`, содержащие информацию о работающем процессе postmaster (а не о том процессе postmaster, который будет восстанавливать эту копию). (Эти файлы могут ввести pg_ctl в заблуждение.)

Часто также стоит исключать из резервной копии каталог `pg_replslot/` кластера, чтобы слоты репликации, существующие на главном сервере, не попадали в копию. В противном случае при последующем восстановлении копии на резервном сервере может получиться так, что он будет неограниченно долго сохранять файлы WAL, а главный не будет очищаться, если он следит за горячим резервом, так как клиенты этих слотов репликации будут продолжать подключаться и изменять состояние слотов на главном, а не резервном сервере. Даже если резервная копия предназначена только для создания нового главного сервера, копирование слотов репликации вряд ли принесёт пользу, так как к моменту включения в работу этого нового сервера содержимое этих слотов станет абсолютно неактуальным.

Содержимое каталогов `pg_dynshmem/`, `pg_notify/`, `pg_serial/`, `pg_snapshots/`, `pg_stat_tmp/` и `pg_subtrans/` (но не сами эти каталоги) можно исключить из резервной копии, так как оно будет инициализировано при запуске главного процесса. Если переменная stats_temp_directory установлена и указывает на подкаталог внутри каталога данных, содержимое этого подкаталога также можно не копировать.

Из резервной копии можно исключить и файлы и подкаталоги с именами, начинающимся с `pgsql_tmp`. Эти файлы удаляются при запуске главного процесса, а каталоги создаются по мере необходимости.

В файл метки резервной копии записывается строка метки, заданная при вызове `pg_start_backup`, время запуска функции `pg_start_backup` и имя начального файла WAL. Таким образом, в случае сомнений можно заглянуть внутрь архива резервной копии и точно определить, в каком сеансе резервного копирования он был создан. Файл карты табличных пространств содержит имена символических ссылок, как они существуют в каталоге `pg_tblspc/`, и полный путь каждой символической ссылки. Эти файлы не только к вашему сведению; их существование и содержание важны для правильного проведения процесса восстановления системы.

Вы также можете создать резервную копию, когда сервер остановлен. В этом случае, вы, очевидно, не сможете вызвать `pg_start_backup` или `pg_stop_backup`, и следовательно, вам надо будет самостоятельно как-то идентифицировать резервные копии и понимать, какие файлы WAL должны быть заархивированы. Поэтому обычно всё-таки лучше следовать вышеописанной процедуре непрерывного архивирования.

24.3.4. Восстановление непрерывной архивной копии

Допустим, худшее случилось, и вам необходимо восстановить базу данных из резервной копии. Порядок действий таков:

1. Остановите сервер баз данных, если он запущен.
2. Если у вас есть место для этого, скопируйте весь текущий каталог кластера баз данных и все табличные пространства во временный каталог на случай, если они вам понадобятся. Учтите,

что эта мера предосторожности требует, чтобы свободного места на диске было достаточно для размещения двух копий существующих данных. Если места недостаточно, необходимо сохранить как минимум содержимое подкаталога `pg_wal` каталога кластера, так как он может содержать журналы, не попавшие в архив перед остановкой системы.

3. Удалите все существующие файлы и подкаталоги из каталога кластера и из корневых каталогов используемых табличных пространств.
4. Восстановите файлы базы данных из резервной копии файлов. Важно, чтобы у восстановленных файлов были правильные разрешения и правильный владелец (пользователь, запускающий сервер, а не `root`!). Если вы используете табличные пространства, убедитесь также, что символьные ссылки в `pg_tblspc/` восстановились корректно.
5. Удалите все файлы из `pg_wal/`; они восстановились из резервной копии файлов и поэтому, скорее всего, будут старее текущих. Если вы вовсе не архивировали `pg_wal/`, создайте этот каталог с правильными правами доступа, но если это была символьная ссылка, восстановите её.
6. Если на шаге 2 вы сохранили незаархивированные файлы с сегментами WAL, скопируйте их в `pg_wal/`. (Лучше всего именно копировать, а не перемещать их, чтобы у вас остались неизменённые файлы на случай, если возникнет проблема и всё придётся начинать сначала.)
7. Создайте командный файл восстановления `recovery.conf` в каталоге кластера баз данных (см. Главу 26). Вы можете также временно изменить `pg_hba.conf`, чтобы обычные пользователи не могли подключаться, пока вы не будете уверены, что восстановление завершилось успешно.
8. Запустите сервер. Сервер запустится в режиме восстановления и начнёт считывать необходимые ему архивные файлы WAL. Если восстановление будет прервано из-за внешней ошибки, сервер можно просто перезапустить и он продолжит восстановление. По завершении процесса восстановления сервер переименует файл `recovery.conf` в `recovery.done` (чтобы предотвратить повторный запуск режима восстановления), а затем перейдёт к обычной работе с базой данных.
9. Просмотрите содержимое базы данных, чтобы убедиться, что вы вернули её к желаемому состоянию. Если это не так, вернитесь к шагу 1. Если всё хорошо, разрешите пользователям подключаться к серверу, восстановив обычный файл `pg_hba.conf`.

Ключевой момент этой процедуры заключается в создании файла конфигурации восстановления, описывающего, как будет выполняться восстановление и до какой точки. В качестве прототипа вы можете использовать файл `recovery.conf.sample` (он обычно помещается в каталог `share/` после установки). Единственное, что совершенно необходимо указать в `recovery.conf` — это команду `restore_command`, которая говорит Postgres Pro, как получать из архива файл-сегменты WAL. Как и `archive_command`, это командная строка для оболочки. Она может содержать символы `%f`, которые заменятся именем требующегося файла журнала, и `%p`, которые заменятся целевым путём для копирования этого файла. (Путь задаётся относительно текущего рабочего каталога, т. е. каталога кластера данных.) Если вам нужно включить в команду сам символ `%`, напишите `%%`. Простейшая команда, которая может быть полезна, такая:

```
restore_command = 'cp /mnt/server/archivedir/%f %p'
```

Эта команда копирует заархивированные ранее сегменты WAL из каталога `/mnt/server/archivedir`. Разумеется, вы можете использовать что-то более сложное, возможно, даже скрипт оболочки, который укажет оператору установить соответствующую ленту.

Важно, чтобы данная команда возвращала ненулевой код возврата в случае ошибки. Эта команда *будет* вызываться и с запросом файлов, отсутствующих в архиве; в этом случае она должна вернуть ненулевое значение и это считается штатной ситуацией. В исключительной ситуации, когда команда была прервана сигналом (кроме SIGTERM, который применяется в процессе остановки сервера базы данных) или произошла ошибка оболочки (например, команда не найдена), восстановление будет прервано и сервер не запустится.

Не все запрашиваемые файлы будут сегментами WAL; следует также ожидать запросов файлов с суффиксом `.history`. Также учтите, что базовое имя пути `%p` будет отличаться от `%f`; не думайте, что они взаимозаменяемы.

Сегменты WAL, которые не найдутся в архиве, система будет искать в `pg_wal/`; благодаря этому можно использовать последние незаархивированные сегменты. Однако файлы в `pg_wal/` будут менее предпочтительными, если такие сегменты окажутся в архиве.

Обычно при восстановлении обрабатываются все доступные сегменты WAL и, таким образом, база данных восстанавливается до последнего момента времени (или максимально близкого к нему, в зависимости от наличия сегментов WAL). Таким образом, восстановление обычно завершается с сообщением «файл не найден»; точный текст сообщения об ошибке зависит от того, что делает `restore_command`. Вы также можете увидеть сообщение об ошибке в начале восстановления для файла с именем типа `00000001.history`. Это также нормально и обычно не говорит о какой-либо проблеме при восстановлении в простых ситуациях; подробнее об этом рассказывается в Подразделе 24.3.5.

Если вы хотите восстановить базу на какой-то момент времени (скажем, до момента, когда неопытный администратор базы данных удалил основную таблицу транзакций), просто укажите требуемую точку остановки в `recovery.conf`. Вы можете задать эту точку, иначе называемую «целью восстановления», по дате/времени, именованной точке восстановления или определённому идентификатору транзакции. На момент написания этой документации полезными могут быть только указания даты/времени или имени точки восстановления, пока нет никаких средств, позволяющих точно определить, какой идентификатор транзакции нужно выбрать.

> **Примечание**
>
> Точка останова должна указывать на момент после окончания базового копирования, т. е. после времени завершения `pg_stop_backup`. Использовать базовую резервную копию для восстановления на момент времени, когда она ещё только создавалась, нельзя. (Чтобы восстановить данные на этот момент времени, придётся вернуться к предыдущей базовой резервной копии и накатывать изменения с этой позиции.)

Если при восстановлении обнаруживаются повреждённые данные WAL, восстановление прерывается в этом месте и сервер не запускается. В этом случае процесс восстановления можно перезапустить с начала, указав «цель восстановления» до точки повреждения, чтобы восстановление могло завершиться нормально. Если восстановление завершается ошибкой из-за внешней причины, например, из-за краха системы или недоступности архива WAL, его можно просто перезапустить, и оно продолжится с того места, где было прервано. Перезапуск восстановления реализован по тому же принципу, что и контрольные точки при обычной работе: сервер периодически сохраняет всё текущее состояние на диске и отражает это в файле `pg_control`, чтобы уже обработанные данные WAL не приходилось сканировать снова.

24.3.5. Линии времени

Возможность восстановить базу данных на некий предыдущий момент времени создаёт некоторые сложности, сродни научно-фантастическим историям о путешествиях во времени и параллельных мирах. Например, предположим, что в начальной истории базы данных вы удалили важную таблицу в 17:15 во вторник, но осознали эту ошибку только в среду в полдень. Вы можете спокойно взять резервную копию, восстановить данные на 17:14 во вторник и запустить сервер. В *этой* истории мира базы данных вы никогда не удаляли вышеупомянутую таблицу. Но предположим, что позже вы заметили, что это была не такая уж хорошая идея и захотели вернуться к утру среды в первоначальной истории базы данных. Вы не сможете сделать это, если в процессе работы базы данных она успеет перезаписать какие-либо файлы-сегменты WAL, приводящие к моменту времени, к которому вы хотите вернуться теперь. Таким образом, для получения желаемого результата необходимо как-то отличать последовательности записей WAL, добавленные после восстановления на какой-то момент времени от тех, что существовали в начальной истории базы данных.

Для решения этой проблемы в Postgres Pro есть такое понятие, как *линия времени*. Всякий раз, когда завершается восстановление из архива, создаётся новая линия времени, позволяющая

идентифицировать последовательность записей WAL, добавленных после этого восстановления. Номер линии времени включается в имя файлов-сегментов WAL, так что файлы новой линии времени не перезаписывают файлы WAL, сгенерированные предыдущими линиями времени. Фактически это позволяет архивировать много различных линий времени. Хотя это может показаться бесполезной возможностью, на самом деле она часто бывает спасительной. Представьте, что вы не определились, какую точку времени выбрать для восстановления, и таким образом должны проводить восстановление методом проб и ошибок, пока не найдёте лучший момент для ответвления от старой истории. Без линий времени этот процесс быстро стал бы очень запутанным. А благодаря линиям времени, вы можете вернуться к *любому* предыдущему состоянию, включая состояния в ветках линий времени, покинутых ранее.

Каждый раз, когда образуется новая линия времени, Postgres Pro создаёт файл «истории линии времени», показывающий, от какой линии времени ответвилась данная и когда. Эти файлы истории нужны, чтобы система могла выбрать правильные файлы-сегменты WAL при восстановлении из архива, содержащего несколько линий времени. Таким образом, они помещаются в область архивов WAL так же, как и файлы сегментов WAL. Файлы истории представляют собой небольшие текстовые файлы, так что они не занимают много места и их вполне можно сохранять неограниченно долго (в отличие от файлов сегментов, имеющих большой размер). Если хотите, вы можете добавлять в файл истории комментарии, свои собственные заметки о том, как и почему была создана эта конкретная линия времени. Такие комментарии будут особенно ценны, если в результате экспериментов у вас образуется хитросплетение разных линий времени.

По умолчанию при восстановлении восстанавливается та же линия времени, которая была текущей при создании базовой резервной копии. Если вы хотите восстановить состояние на какой-либо дочерней линии времени, (то есть, хотите вернуться к некоторому состоянию, которое тоже было получено в результате попытки восстановления), вам необходимо указать идентификатор целевой линии времени в `recovery.conf`. Восстановить состояние в линии времени, ответившейся раньше, чем была сделана базовая резервная копия, нельзя.

24.3.6. Советы и примеры

Ниже мы дадим несколько советов по настройке непрерывного архивирования.

24.3.6.1. Обособленные горячие резервные копии

Средства резервного копирования Postgres Pro можно применять для создания обособленных горячих копий. Эти копии нельзя использовать для восстановления на момент времени, но создаются и восстанавливаются они обычно гораздо быстрее, чем дампы pg_dump. (Они также намного больше, чем дампы pg_dump, так что в некоторых случаях выигрыш в скорости может быть потерян.)

Как и базовые резервные копии, обособленную горячую копию проще всего получить, используя программу pg_basebackup. Если вы вызовете эту программу с параметром `-X`, в эту копию автоматически будет включён весь журнал предзаписи, необходимый для её использования, так что никакие особые действия для восстановления не потребуются.

Если нужна дополнительная гибкость в процессе копирования файлов, создавать обособленные горячие копии можно также на более низком уровне. Чтобы подготовиться к получению такой копии на низком уровне, установите в `wal_level` уровень `replica` (или выше), в `archive_mode` значение `on` и настройте команду `archive_command`, которая будет выполнять архивацию, только когда существует *файл-переключатель*. Например:

```
archive_command = 'test ! -f /var/lib/pgsql/backup_in_progress || (test ! -f /var/lib/
pgsql/archive/%f && cp %p /var/lib/pgsql/archive/%f)'
```

Данная команда выполнит архивацию, если будет существовать файл `/var/lib/pgsql/backup_in_progress`, а в противном случае просто вернёт нулевой код возврата (и тогда Postgres Pro сможет переработать ненужный файл WAL).

После такой подготовки резервную копию можно создать, например таким скриптом:

```
touch /var/lib/pgsql/backup_in_progress
psql -c "select pg_start_backup('hot_backup');"
tar -cf /var/lib/pgsql/backup.tar /var/lib/pgsql/data/
psql -c "select pg_stop_backup();"
rm /var/lib/pgsql/backup_in_progress
tar -rf /var/lib/pgsql/backup.tar /var/lib/pgsql/archive/
```

Сначала создаётся файл-переключатель `/var/lib/pgsql/backup_in_progress`, включающий архивирование заполненных файлов WAL. По окончании резервного копирования файл-переключатель удаляется. Затем заархивированные файлы WAL тоже добавляются в резервную копию, так что в одном архиве tar оказывается и базовая резервная копия, и все требуемые файлы WAL. Пожалуйста, не забудьте добавить в ваши скрипты резервного копирования обработку ошибок.

24.3.6.2. Сжатие журналов в архиве

Если размер архива имеет большое значение, можно воспользоваться gzip и сжимать архивные файлы:

```
archive_command = 'gzip < %p > /var/lib/pgsql/archive/%f'
```

При этом для восстановления придётся использовать gunzip:

```
restore_command = 'gunzip < /mnt/server/archivedir/%f > %p'
```

24.3.6.3. Скрипты `archive_command`

Многие в качестве команды `archive_command` используют скрипты, так что запись в `postgresql.conf` оказывается очень простой:

```
archive_command = 'local_backup_script.sh "%p" "%f"'
```

Применять отдельный файла скрипта целесообразно всегда, когда вы хотите использовать в процедуре архивирования несколько команд. Это позволяет управлять сложностью этой процедуры в рамках одного скрипта, который можно написать на любом популярном языке скриптов, например на bash или perl.

В частности, с помощью скриптов можно решить такие задачи:

- Копирование данных в безопасное внешнее хранилище
- Пакетная обработка файлов WAL, чтобы они передавались каждые три часа, а не по одному
- Взаимодействие с другими приложениями резервного копирования и восстановления
- Взаимодействие со средствами мониторинга, регистрация ошибок

> **Подсказка**
>
> Когда в `archive_command` используется скрипт, желательно включить logging_collector. Тогда все сообщения, которые скрипт выведет в stderr, будут записываться в журнал сервера баз данных, что позволит легко диагностировать ошибки в сложных конфигурациях.

24.3.7. Ограничения

На момент написания документации методика непрерывного архивирования имеет несколько ограничений. Они могут быть ликвидированы в будущих версиях:

- Если во время создания базовой резервной копии выполняется команда CREATE DATABASE, а затем база-шаблон, задействованная в `CREATE DATABASE`, изменяется, пока продолжается копирование, возможно, что при восстановлении эти изменения распространятся также и на созданную базу данных. Конечно, это нежелательно. Во избежание подобных рисков, лучше всего не изменять никакие базы-шаблоны во время получения базовой резервной копии.

- Команды CREATE TABLESPACE записываются в WAL с абсолютным путём и, таким образом, при воспроизведении WAL будут выполнены с тем же абсолютным путём. Это может быть нежелательно, если журнал воспроизводится на другой машине. Но опасность есть, даже если журнал воспроизводится на той же машине, но в другом каталоге данных: при воспроизведении будет так же перезаписано содержимое исходных табличных пространств. Чтобы избежать потенциальных проблем такого рода, лучше всего делать новую базовую резервную копию после создания или удаления табличных пространств.

Также следует заметить, что стандартный формат WAL не очень компактный, так как включает много снимков дисковых страниц. Эти снимки страниц предназначены для поддержки восстановления после сбоя, на случай, если понадобится исправить страницы, записанные на диск частично. В зависимости от аппаратного и программного обеспечения вашей системы, риск частичной записи может быть достаточно мал, так что его можно игнорировать, и в этом случае можно существенно уменьшить общий объём архивируемых журналов, выключив снимки страниц с помощью параметра full_page_writes. (Прежде чем делать это, прочтите замечания и предупреждения в Главе 29.) Выключение снимков страниц не препятствует использованию журналов для восстановления PITR. Одним из направлений разработки в будущем является сжатие архивируемых данных WAL, путём удаления ненужных копий страниц даже при включённом режиме `full_page_writes`. Тем временем администраторы могут сократить количество снимков страниц, включаемых в WAL, увеличив параметры интервала контрольных точек в разумных пределах.

Глава 25. Отказоустойчивость, балансировка нагрузки и репликация

Серверы базы данных могут работать совместно для обеспечения возможности быстрого переключения на другой сервер в случае отказа первого (отказоустойчивость) или для обеспечения возможности нескольким серверам БД обрабатывать один набор данных (балансировка нагрузки). В идеале, серверы БД могут работать вместе прозрачно для клиента. Веб-серверы, обрабатывающие статические страницы, можно совместить достаточно легко посредством простого распределения запросов на несколько машин. Фактически серверы баз данных только для чтения тоже могут быть совмещены достаточно легко. К сожалению, большинство серверов баз данных получают смешанные запросы на чтение/запись, а серверы с доступом на чтение/запись совместить гораздо сложнее. Это объясняется тем, что данные только для чтения достаточно единожды разместить на каждом сервере, а запись на любой из серверов должна распространиться на все остальные серверы, чтобы будущие запросы на чтение возвращали согласованные результаты.

Проблема синхронизации является главным препятствием для совместной работы серверов. Так как единственного решения, устраняющего проблему синхронизации во всех случаях, не существует, предлагается несколько решений. Разные решения подходят к проблеме по-разному и минимизируют её влияние в разных рабочих условиях.

Некоторые решения применяют синхронизацию, позволяя только одному серверу изменять данные. Сервер, который может изменять данные, называется сервером чтения/записи, *ведущим* или *главным* сервером. Сервер, который отслеживает изменения на ведущем, называется *ведомым* или *резервным* сервером. Резервный сервер, к которому нельзя подключаться до тех пор, пока он не будет повышен до главного, называется сервером *тёплого резерва*, а тот, который может принимать соединения и обрабатывать запросы только на чтение, называется сервером *горячего резерва*.

Некоторые решения являются синхронными, при которых транзакция, модифицирующая данные, не считается подтверждённой, пока все серверы не подтвердят транзакцию. Это гарантирует, что при отработке отказа не произойдёт потеря данных и что все балансирующие серверы возвращают целостные данные вне зависимости от того, к какому серверу был запрос. Асинхронное решение, напротив, допускает некоторую задержку между временем подтверждения транзакции и её передачей на другие серверы, допуская возможность, что некоторые транзакции могут быть потеряны в момент переключения на резервный сервер и что балансирующие серверы могут вернуть слегка устаревшие данные. Асинхронная передача используется, когда синхронная будет слишком медленной.

Решения могут так же разделяться по степени детализации. Некоторые решения работают только на уровне всего сервера БД целиком, в то время как другие позволяют работать на уровне таблиц или уровне БД.

В любом случае необходимо принимать во внимание быстродействие. Обычно выбирается компромисс между функциональностью и производительностью. Например, полностью синхронное решение в медленной сети может снизить производительность больше чем наполовину, в то время как асинхронное решение будет оказывать минимальное воздействие.

В продолжении этого раздела рассматриваются различные решения по организации отказоустойчивости, репликации и балансировки нагрузки.

25.1. Сравнение различных решений

Отказоустойчивость на разделяемых дисках

Отказоустойчивость на разделяемых дисках позволяет избежать избыточности синхронизации путём задействования только одной копии базы данных. Она использует единственный дисковый массив, который разделяется между несколькими серверами. Если основной сервер

БД откажет, резервный сервер может подключиться и запустить базу данных, что позволит восстановить БД после аварии. Это обеспечивает быстрое переключение без потери данных.

Функциональность разделяемого оборудования обычно реализована в сетевых устройствах хранения. Так же возможно применение сетевой файловой системы; особое внимание следует уделить тому, чтобы поведение системы полностью соответствовало POSIX (см. Подраздел 17.2.2). Существенное ограничение этого метода состоит в том, что в случае отказа или порчи разделяемого дискового массива оба сервера: главный и резервный — станут нерабочими. Другая особенность — резервный сервер никогда не получает доступ к разделяемым дискам во время работы главного.

Репликация на уровне файловой системы (блочного устройства)

Видоизменённая версия функциональности разделяемого оборудования представлена в виде репликации на уровне файловой системы, когда все изменения в файловой системе отражаются в файловой системе другого компьютера. Единственное ограничение: синхронизация должна выполняться методом, гарантирующим целостность копии файловой системы на резервном сервере — в частности, запись на резервном сервере должна происходить в том же порядке, что и на главном. DRBD является популярным решением на основе репликации файловой системы для Linux.

Трансляция журнала предзаписи

Серверы тёплого и горячего резерва могут так же поддерживаться актуальными путём чтения потока записей из журнала изменений (WAL). Если основной сервер отказывает, резервный содержит почти все данные с него и может быть быстро преобразован в новый главный сервер БД. Это можно сделать синхронно или асинхронно, но может быть выполнено только на уровне сервера БД целиком.

Резервный сервер может быть реализован с применением трансляции файлов журналов (см. Раздел 25.2), или потоковой репликации (см. Подраздел 25.2.5), или их комбинацией. За информацией о горячем резерве обратитесь к Разделу 25.5.

Логическая репликация

В схеме с логической репликацией сервер баз данных может передавать поток изменений данных на другой сервер. Механизм логической репликации в Postgres Pro формирует поток логических изменений данных, обрабатывая WAL. Логическая репликация позволяет переносить изменения, происходящие только в отдельных таблицах. Для логической репликации не требуется, чтобы за определённым сервером закреплялась роль главного или реплицирующего; напротив, данные могут передаваться в разных направлениях. За дополнительными сведениями о логической репликации обратитесь к Главе 30. Используя интерфейс логического декодирования (Глава 46), подобную функциональность могут предоставлять и сторонние расширения.

Репликация главный-резервный на основе триггеров

При репликации главный-резервный все запросы, изменяющие данные, пересылаются главному серверу. Главный сервер, в свою очередь, асинхронно пересылает изменённые данные резервному. Резервный сервер может обрабатывать запросы только на чтение при работающем главном. Такой резервный сервер идеален для обработки запросов к хранилищам данных.

Slony-I является примером подобного типа репликации, действующей на уровне таблиц, и поддерживает множество резервных серверов. Так как обновления на резервных серверах происходят асинхронно (в пакетах), возможна потеря данных во время отказа.

Репликация запросов в среднем слое

В схеме с репликацией запросов в среднем слое, средний слой перехватывает каждый SQL-запрос и пересылает его на один или все серверы. Каждый сервер работает независимо. Модифицирующие запросы должны быть направлены всем серверам, чтобы каждый из них получал все изменения. Но читающие запросы могут посылаться только одному серверу, что позволяет перераспределить читающую нагрузку между всеми серверами.

Если запросы просто перенаправлять без изменений, функции подобные `random()`, `CURRENT_TIMESTAMP` и последовательности могут получить различные значения на разных серверах. Это происходит потому что каждый сервер работает независимо, а эти запросы неизбирательные (и действительно не изменяют строки). Если такая ситуация недопустима, или средний слой, или приложение должны запросить подобные значения с одного сервера, затем использовать его в других пишущих запросах. Другим способом является применения этого вида репликации совместно с другим традиционным набором репликации главный-резервный, то есть изменяющие данные запросы посылаются только на главный сервер, а затем применяются на резервном в процессе этой репликации, но не с помощью реплицирующего среднего слоя. Следует иметь в виду, что все транзакции фиксируются или прерываются на всех серверах, возможно с применением двухфазной фиксации (см. PREPARE TRANSACTION и COMMIT PREPARED). Репликацию такого типа реализуют, например Pgpool-II и Continuent Tungsten.

Асинхронная репликация с несколькими ведущими

Если серверы не находятся постоянно в единой сети или связаны низкоскоростным каналом, как например, ноутбуки или удалённые серверы, обеспечение согласованности данных между ними представляет проблему. Когда используется асинхронная репликация с несколькими ведущими серверами, каждый из них работает независимо и периодически связывается с другими серверами для определения конфликтующих транзакций. Конфликты могут урегулироваться пользователем или по правилам их разрешения. Примером такого типа репликации является Bucardo.

Синхронная репликация с несколькими ведущими

При синхронной репликации с несколькими ведущими серверами каждый сервер может принимать запросы на запись, а изменённые данные передаются с получившего их сервера всем остальным, прежде чем транзакция будет подтверждена. Если запись производится интенсивно, это может провоцировать избыточные блокировки и задержки при фиксации, что приводит к снижению производительности. Запросы на чтение могут быть обработаны любым сервером. В некоторых конфигурациях для более эффективного взаимодействия серверов применяются разделяемые диски. Синхронная репликация с несколькими ведущими лучше всего работает, когда преобладают операции чтения, хотя её большой плюс в том, что любой сервер может принимать запросы на запись — нет необходимости искусственно разделять нагрузку между главным и резервными серверами, а так как изменения передаются от одного сервера другим, не возникает проблем с недетерминированными функциями вроде `random()`.

Postgres Pro Enterprise включает расширение `multimaster`, реализующее этот вид репликации. Используя `multimaster`, вы можете организовать синхронный кластер без разделения ресурсов, который обеспечивает расширяемость OLTP для читающих транзакций, а также высокую степень доступности с автоматическим восстановлением после сбоев.

Таблица 25.1 итоговая таблица возможностей различных решений приведена ниже.

Таблица 25.1. Таблица свойств отказоустойчивости, балансировки нагрузки и репликации

Тип	**Отказоустойчивость через разделяемые диски**	**Репликация файловой системы**	**Трансляция журнала предзаписи**	**Логическая репликация**	**Репликация главный-резервный на основе триггеров**	**Репликация запросов в среднем слое**	**Асинхронная репликация с несколькими главными серверами**	**Синхронная репликация с несколькими главными серверами**
Наиболее типичные реализации	NAS	DRBD	встроенная потоковая репликация	встроенная логическая репликация, pglogical	Londiste, Slony	pgpool-II	Bucardo	расширение `multimaster` в составе Postgres Pro Enterprise

Тип	Отказоустойчивость через разделяемые диски	Репликация файловой системы	Трансляция журнала предзаписи	Логическая репликация	Репликация главный-резервный на основе триггеров	Репликация запросов в среднем слое	Асинхронная репликация с несколькими главными серверами	Синхронная репликация с несколькими главными серверами
Метод взаимодействия	разделяемые диски	дисковые блоки	WAL	логическое декодирование	Строки таблицы	SQL	Строки таблицы	Строки таблицы и блокировки строк
Не требуется специального оборудования		•	•	•	•	•	•	•
Допускается несколько ведущих серверов				•		•	•	•
Нет доп. нагрузки ведущего сервера	•		•	•		•		
Нет задержки при нескольких серверах	•		без синхр.	без синхр.	•		•	
Отказ ведущего сервера не может привести к потере данных	•	•	с синхр.	с синхр.		•		•
Сервер реплики принимает читающие запросы			с горячим резервом	•	•	•	•	•
Репликация на уровне таблиц				•	•		•	•
Не требуется разрешение конфликтов	•	•	•		•	•		•

Несколько решений, которые не подпадают под указанные выше категории:

Секционирование данных

При секционировании таблицы расщепляются на наборы данных. Каждый из наборов может быть изменён только на одном сервере. Например, данные могут быть секционированы по офисам, например, Лондон и Париж, с сервером в каждом офисе. В случае необходимости

обращения одновременно к данным Лондона и Парижа, приложение может запросить оба сервера, или может быть применена репликация главный-резервный для предоставления копии только для чтения в другом офисе для каждого из серверов.

Выполнение параллельных запросов на нескольких серверах

Многие из описанных выше решений позволяют обрабатывать несколько запросов на нескольких серверах, но ни одно из них не поддерживает выполнение одного запроса на нескольких серверах, что позволило бы его ускорить. Данное же решение позволяет нескольким серверам обрабатывать один запрос одновременно. Для этого обычно данные разделяются между серверами, серверы выполняют свои части запросов, выдают результаты центральному серверу, а он, в свою очередь, объединяет полученные данные и выдаёт итоговый результат пользователю. Это решение может быть реализовано с применением набора средств PL/Proxy.

25.2. Трансляция журналов на резервные серверы

Постоянная архивация может использоваться для создания кластерной конфигурации *высокой степени доступности* (HA) с одним или несколькими *резервными серверами*, способными заменить ведущий сервер в случае выхода его из строя. Такую реализацию отказоустойчивости часто называют *тёплый резерв* или *трансляция журналов*.

Ведущий и резервный серверы работают совместно для обеспечения этой возможности, при этом они связаны опосредованно. Ведущий сервер работает в режиме постоянной архивации изменений, в то время как каждый резервный сервер работает в режиме постоянного приёма архивных изменений, получая файлы WAL от ведущего. Для обеспечения этой возможности не требуется вносить изменения в таблицы БД, поэтому администрировать данное решение репликации проще, чем ряд других. Так же такая конфигурация относительно слабо влияет на производительность ведущего сервера.

Непосредственную передачу записей WAL с одного сервера БД на другой обычно называют трансляцией журналов (или доставкой журналов). Postgres Pro реализует трансляцию журналов на уровне файлов, передавая записи WAL по одному файлу (сегменту WAL) единовременно. Файлы WAL (размером 16 МБ) можно легко и эффективно передать на любое расстояние, будь то соседний сервер, другая система в местной сети или сервер на другом краю света. Требуемая пропускная способность при таком подходе определяется скоростью записи транзакций на ведущем сервере. Трансляция журналов на уровне записей более фрагментарная операция, при которой изменения WAL передаются последовательно через сетевое соединение (см. Подраздел 25.2.5).

Следует отметить, что трансляция журналов асинхронна, то есть записи WAL доставляются после завершения транзакции. В результате образуется окно, когда возможна потеря данных при отказе сервера: будут утеряны ещё не переданные транзакции. Размер этого окна при трансляции файлов журналов может быть ограничен параметром `archive_timeout`, который может принимать значение меньше нескольких секунд. Тем не менее подобные заниженные значения могут потребовать существенного увеличения пропускной способности, необходимой для трансляции файлов. При потоковой репликации (см. Подраздел 25.2.5) окно возможности потери данных гораздо меньше.

Скорость восстановления достаточно высока, обычно резервный сервер становится полностью доступным через мгновение после активации. В результате такое решение называется тёплым резервом, что обеспечивает отличную отказоустойчивость. Восстановление сервера из архивной копии базы и применение изменений обычно происходит существенно дольше. Поэтому такие действия обычно требуются при восстановлении после аварии, не для отказоустойчивости. Так же резервный сервер может обрабатывать читающие запросы. В этом случае он называется сервером горячего резерва. См. Раздел 25.5 для подробной информации.

25.2.1. Планирование

Обычно разумно подбирать ведущий и резервный серверы так, чтобы они были максимально похожи, как минимум с точки зрения базы данных. Тогда в частности, пути, связанные с

табличными пространствами, могут передаваться без изменений. Таким образом, как на ведущем, так и на резервных серверах должны быть одинаковые пути монтирования для табличных пространств при использовании этой возможности БД. Учитывайте, что если CREATE TABLESPACE выполнена на ведущем сервере, новая точка монтирования для этой команды уже должна существовать на резервных серверах до её выполнения. Аппаратная часть не должна быть в точности одинаковой, но опыт показывает, что сопровождать идентичные системы легче, чем две различные на протяжении жизненного цикла приложения и системы. В любом случае архитектура оборудования должна быть одинаковой — например, трансляция журналов с 32-битной на 64-битную систему не будет работать.

В общем случае трансляция журналов между серверами с различными основными версиями Postgres Pro невозможна. Политика главной группы разработки Postgres Pro состоит в том, чтобы не вносить изменения в дисковые форматы при обновлениях корректирующей версии, таким образом, ведущий и резервный серверы, имеющие разные корректирующие версии, могут работать успешно. Тем не менее формально такая возможность не поддерживается, поэтому рекомендуется поддерживать одинаковую версию ведущего и резервных серверов, насколько это возможно. При обновлении корректирующей версии безопаснее будет в первую очередь обновить резервные серверы — новая корректирующая версия с большей вероятностью прочитает файл WAL предыдущей корректирующей версии, чем наоборот.

25.2.2. Работа резервного сервера

Сервер, работающий в режиме резервного, последовательно применяет файлы WAL, полученные от главного. Резервный сервер может читать файлы WAL из архива WAL (см. restore_command) или напрямую с главного сервера по соединению TCP (потоковая репликация). Резервный сервер также будет пытаться восстановить любой файл WAL, найденный в кластере резервного в каталоге `pg_wal`. Это обычно происходит после перезапуска сервера, когда он применяет заново файлы WAL, полученные от главного сервера перед перезапуском. Но можно и вручную скопировать файлы в каталог `pg_wal`, чтобы применить их в любой момент времени.

В момент запуска резервный сервер начинает восстанавливать все доступные файлы WAL, размещённые в архивном каталоге, указанном в команде `restore_command`. По достижении конца доступных файлов WAL или при сбое команды `restore_command` сервер пытается восстановить все файлы WAL, доступные в каталоге `pg_wal`. Если это не удаётся и потоковая репликация настроена, резервный сервер пытается присоединиться к ведущему и начать закачивать поток WAL с последней подтверждённой записи, найденной в архиве или `pg_wal`. Если это действие закончилось неудачей, или потоковая репликация не настроена, или соединение позднее разорвалось, резервный сервер возвращается к шагу 1 и пытается восстановить файлы из архива вновь. Цикл обращения за файлами WAL к архиву, `pg_wal`, и через потоковую репликацию продолжается до остановки сервера или переключения его роли, вызванного файлом-триггером.

Режим резерва завершается и сервер переключается в обычный рабочий режим при получении команды `pg_ctl promote` или при обнаружении файла-триггера (`trigger_file`). Перед переключением сервер восстановит все файлы WAL, непосредственно доступные из архива или `pg_wal`, но пытаться подключиться к главному серверу он больше не будет.

25.2.3. Подготовка главного сервера для работы с резервными

Настройка постоянного архивирования на ведущем сервере в архивный каталог, доступный с резервного, описана в Разделе 24.3. Расположение архива должно быть доступно с резервного сервера даже при отключении главного, то есть его следует разместить на резервном или другом доверенном, но не на главном сервере.

При использовании потоковой репликации следует настроить режим аутентификации на ведущем сервере, чтобы разрешить соединения с резервных. Для этого создать роль и обеспечить подходящую запись в файле `pg_hba.conf` в разделе доступа к БД `replication`. Так же следует убедиться, что для параметра `max_wal_senders` задаётся достаточно большое значение в конфигурационном файле ведущего сервера. При использовании слотов для репликации также достаточно большое значение нужно задать для `max_replication_slots`.

Создание базовой резервной копии, необходимой для запуска резервного сервера, описано в Подразделе 24.3.2.

25.2.4. Настройка резервного сервера

Для запуска резервного сервера нужно восстановить резервную копию, снятую с ведущего (см. Подраздел 24.3.4). Затем нужно создать файл команд восстановления `recovery.conf` в каталоге данных кластера резервного сервера и включить режим `standby_mode`. Задайте в `restore_command` обычную команду копирования файлов из архива WAL. Если планируется несколько резервных серверов в целях отказоустойчивости, установите для `recovery_target_timeline` значение `latest`, чтобы резервный сервер переходил на новую линию времени, образуемую при отработке отказа и переключении на другой сервер.

> **Примечание**
>
> Не используйте pg_standby или подобные средства со встроенным режимом резервного сервера, описанным здесь. Команда `restore_command` должна немедленно прекратиться при отсутствии файла; сервер повторит команду вновь при необходимости. Использование средств, подобных pg_standby, описано в Разделе 25.4.

При необходимости потоковой репликации заполните `primary_conninfo` параметрами строки соединения для libpq, включая имя (или IP-адрес) сервера и все остальные необходимые данные для соединения с ведущим сервером. Если ведущий требует пароль для аутентификации, пароль также должен быть указан в `primary_conninfo`.

Если резервный сервер настраивается в целях отказоустойчивости, на нём следует настроить архивацию WAL, соединения и аутентификацию, как на ведущем сервере, потому что резервный сервер станет ведущим после отработки отказа.

При использовании архива WAL его размер может быть уменьшен с помощью команды в параметре archive_cleanup_command, которая удаляет файлы уже не нужные для дальнейшей работы резервного сервера. Утилита pg_archivecleanup разработана специально для использования в `archive_cleanup_command` при типичной конфигурации с одним резервным сервером (см. pg_archivecleanup). Следует отметить, что если архив используется в целях резервирования, следует сохранять все файлы необходимые для восстановления как минимум с последней базовой резервной копии, даже если они не нужны для резервного сервера.

Простой пример `recovery.conf`:

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
restore_command = 'cp /path/to/archive/%f %p'
archive_cleanup_command = 'pg_archivecleanup /path/to/archive %r'
```

Можно поддерживать любое количество резервных серверов, но при применении потоковой репликации необходимо убедиться, что значение `max_wal_senders` на ведущем достаточно большое, чтобы все они могли подключиться одновременно.

25.2.5. Потоковая репликация

При потоковой репликации резервный сервер может работать с меньшей задержкой, чем при трансляции файлов. Резервный сервер подключается к ведущему, который передаёт поток записей WAL резервному в момент их добавления, не дожидаясь окончания заполнения файла WAL.

Потоковая репликация асинхронна по умолчанию (см. Подраздел 25.2.8), то есть имеется небольшая задержка между подтверждением транзакции на ведущем сервере и появлением этих изменений на резервном. Тем не менее эта задержка гораздо меньше, чем при трансляции файлов журналов, обычно в пределах одной секунды, если резервный сервер достаточно

мощный и справляется с нагрузкой. При потоковой репликации настраивать `archive_timeout` для уменьшения окна потенциальной потери данных не требуется.

При потоковой репликации без постоянной архивации на уровне файлов, сервер может избавиться от старых сегментов WAL до того, как резервный получит их. В этом случае резервный сервер потребует повторной инициализации из новой базовой резервной копии. Этого можно избежать, установив для `wal_keep_segments` достаточно большое значение, при котором сегменты WAL будут защищены от ранней очистки, либо настроив слот репликации для резервного сервера. Если с резервного сервера доступен архив WAL, этого не требуется, так как резервный может всегда обратиться к архиву для восполнения пропущенных сегментов.

Чтобы включить потоковую репликацию, сначала настройте резервный сервер на приём трансляции журналов, как описано в Разделе 25.2. Затем сделайте следующий шаг — переключите резервный сервер в режим репликации, установив в `primary_conninfo` в файле `recovery.conf` строку подключения, указывающую на ведущий. Настройте listen_addresses и параметры аутентификации (см. `pg_hba.conf`) на ведущем сервере таким образом, чтобы резервный смог подключиться к псевдобазе `replication` на ведущем (см. Подраздел 25.2.5.1).

В системах, поддерживающих параметр сокета keepalive, подходящие значения tcp_keepalives_idle, tcp_keepalives_interval и tcp_keepalives_count помогут ведущему вовремя заметить разрыв соединения.

Установите максимальное количество одновременных соединений с резервных серверов (см. описание max_wal_senders).

При запуске резервного сервера с правильно установленным `primary_conninfo` резервный подключится к ведущему после воспроизведения всех файлов WAL, доступных из архива. При успешном установлении соединения можно увидеть процесс walreceiver на резервном сервере и соответствующий процесс walsender на ведущем.

25.2.5.1. Аутентификация

Право использования репликации очень важно ограничить так, чтобы только доверенные пользователи могли читать поток WAL, так как из него можно извлечь конфиденциальную информацию. Резервный сервер должен аутентифицироваться на ведущем от имени суперпользователя или пользователя с правом `REPLICATION`. Настоятельно рекомендуется создавать выделенного пользователя с правами `REPLICATION` и `LOGIN` специально для репликации. Хотя право `REPLICATION` даёт очень широкие полномочия, оно не позволяет модифицировать данные в ведущей системе, тогда как с правом `SUPERUSER` это можно делать.

Список аутентификации клиентов для репликации содержится в `pg_hba.conf` в записях с установленным значением `replication` в поле *database*. Например, если резервный сервер запущен на компьютере с IP-адресом `192.168.1.100` и учётная запись для репликации `foo`, администратор может добавить следующую строку в файл `pg_hba.conf` ведущего:

```
# Разрешить пользователю "foo" с компьютера 192.168.1.100 подключаться к этому
# серверу в качестве партнёра репликации, если был передан правильный пароль.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    replication     foo             192.168.1.100/32        md5
```

Имя компьютера и номер порта для ведущего, имя пользователя для соединения и пароль указываются в файле `recovery.conf`. Пароль также может быть задан в файле `~/.pgpass` на резервном сервере (указанном в определении с `replication` в поле *database*). Например, если ведущий принимает подключения по IP-адресу `192.168.1.50`, через порт `5432`, пользователя для репликации `foo` с паролем `foopass`, то администратор может добавить следующую строку в файл `recovery.conf` на резервном сервере:

```
# Резервный сервер подключается к ведущему, работающему на компьютере 192.168.1.50
# (порт 5432), от имени пользователя "foo" с паролем "foopass".
```

```
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
```

25.2.5.2. Наблюдение

Важным индикатором стабильности работы потоковой репликации является количество записей WAL, созданных на ведущем, но ещё не применённых на резервном сервере. Задержку можно подсчитать, сравнив текущую позиции записи WAL на ведущем с последней позицией WAL, полученной на резервном сервере. Эти позиции можно узнать, воспользовавшись функциями `pg_current_wal_lsn` на ведущем и `pg_last_wal_receive_lsn` на резервном, соответственно (за подробностями обратитесь к Таблице 9.79 и Таблице 9.80). Последняя полученная позиция WAL на резервном сервере также выводится в состоянии процесса-приёмника WAL, которое показывает команда `ps` (подробнее об этом в Разделе 27.1).

Список процессов-передатчиков WAL можно получить через представление `pg_stat_replication`. Большая разница между `pg_current_wal_lsn` и полем `sent_lsn` этого представления может указывать на то, что главный сервер работает с большой нагрузкой, тогда как разница между `sent_lsn` и `pg_last_wal_receive_lsn` на резервном может быть признаком задержек в сети или большой нагрузки резервного сервера.

25.2.6. Слоты репликации

Слоты репликации автоматически обеспечивают механизм сохранения сегментов WAL, пока они не будут получены всеми резервными и главный сервер не будет удалять строки, находящиеся в статусе recovery conflict даже при отключении резервного.

Вместо использования слотов репликации для предотвращения удаления старых сегментов WAL можно применять wal_keep_segments, или сохранять сегменты в архиве с помощью команды archive_command. Тем не менее эти методы часто приводят к тому, что хранится больше сегментов WAL, чем необходимо, в то время как слоты репликации оставляют только то количество сегментов, которое необходимо. Преимущество этих методов состоит в том, что они чётко задают объёмы места, необходимого для `pg_wal`; в то время как текущая реализация слотов репликации не предоставляет такой возможности.

Подобным образом, параметры hot_standby_feedback и vacuum_defer_cleanup_age позволяют защитить востребованные строки от удаления при очистке, но первый параметр не защищает в тот промежуток времени, когда резервный сервер не подключён, а для последнего часто нужно задавать большое значение, чтобы обеспечить должную защиту. Слоты репликации решают эти проблемы.

25.2.6.1. Запросы и действия слотов репликации

Каждый слот репликации обладает именем, состоящим из строчных букв, цифр и символов подчёркивания.

Имеющиеся слоты репликации и их статус можно просмотреть в представлении `pg_replication_slots`.

Слоты могут быть созданы и удалены как с помощью протокола потоковой репликации (см. Раздел 50.4), так и посредством функций SQL (см. Подраздел 9.26.6).

25.2.6.2. Пример конфигурации

Для создания слота репликации выполните:

```
postgres=# SELECT * FROM pg_create_physical_replication_slot('node_a_slot');
  slot_name  | lsn
-------------+-----
 node_a_slot |

postgres=# SELECT slot_name, slot_type, active FROM pg_replication_slots;
  slot_name  | slot_type | active
-------------+-----------+--------
```

```
 node_a_slot | physical  | f
(1 row)
```

Для настройки резервного сервера на использование этого слота `primary_slot_name` должно быть настроено в конфигурации `recovery.conf` резервного. Вот простейший пример:

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
primary_slot_name = 'node_a_slot'
```

25.2.7. Каскадная репликация

Свойство каскадной репликации позволяет резервному серверу принимать соединения репликации и потоки WAL от других резервных, выступающих посредниками. Это может быть полезно для уменьшения числа непосредственных подключений к главному серверу, а также для уменьшения накладных расходов при передаче данных в интрасети.

Резервный сервер, выступающий как получатель и отправитель, называется каскадным резервным сервером. Резервные серверы, стоящие ближе к главному, называются серверами верхнего уровня, а более отдалённые — серверами нижнего уровня. Каскадная репликация не накладывает ограничений на количество или организацию последующих уровней, а каждый резервный соединяется только с одним сервером вышестоящего уровня, который в конце концов соединяется с единственным главным/ведущим сервером.

Резервный сервер каскадной репликации не только получает записи WAL от главного, но так же восстанавливает их из архива. Таким образом, даже если соединение с сервером более высокого уровня разорвётся, потоковая репликация для последующих уровней будет продолжаться до исчерпания доступных записей WAL.

Каскадная репликация в текущей реализации асинхронна. Параметры синхронной репликации (см. Подраздел 25.2.8) в настоящее время не оказывают влияние на каскадную репликацию.

Распространение обратной связи горячего резерва работает от нижестоящего уровня к вышестоящему уровню вне зависимости от способа организации связи.

Если вышестоящий резервный сервер будет преобразован в новый главный, нижестоящие серверы продолжат получать поток с нового главного при условии, что `recovery_target_timeline` установлен в значение `'latest'`.

Для использования каскадной репликации необходимо настроить резервный каскадный сервер на приём соединений репликации (то есть установить max_wal_senders и hot_standby, настроить host-based authentication). Так же может быть необходимо настроить на нижестоящем резервном значение `primary_conninfo` на каскадный резервный сервер.

25.2.8. Синхронная репликация

По умолчанию в Postgres Pro потоковая репликация асинхронна. Если ведущий сервер выходит из строя, некоторые транзакции, которые были подтверждены, но не переданы на резервный, могут быть потеряны. Объём потерянных данных пропорционален задержке репликации на момент отработки отказа.

Синхронная репликация предоставляет возможность гарантировать, что все изменения, внесённые в транзакции, были переданы одному или нескольким синхронным резервным серверам. Это повышает стандартный уровень надёжности, гарантируемый при фиксации транзакции. Этот уровень защиты соответствует второму уровню безопасности репликации из теории вычислительной техники, или групповой безопасности первого уровня (безопасности групповой и уровня 1), когда выбран режим `synchronous_commit remote_write`.

При синхронной репликации каждая фиксация пишущей транзакции ожидает подтверждения того, что запись фиксации помещена в журнал предзаписи на диске на обоих серверах: ведущем и резервном. При таком варианте потеря данных может произойти только в случае

одновременного выхода из строя ведущего и резервного серверов. Это обеспечивает более высокий уровень надёжности, при условии продуманного подхода системного администратора к вопросам размещения и управления этими серверами. Ожидание подтверждения увеличивает уверенность в том, что данные не будут потеряны во время сбоя сервера, но при этом увеличивает время отклика для обработки транзакции. Минимальное время ожидания равно времени передачи данных от ведущего к резервному и обратно.

Транзакции только для чтения и откат транзакции не требуют ожидания для ответа с резервного сервера. Промежуточные подтверждения не ожидают ответа от резервного сервера, только подтверждение верхнего уровня. Долгие операции вида загрузки данных или построения индекса не ожидают финального подтверждения. Но все двухфазные подтверждения требуют ожидания, включая подготовку и непосредственно подтверждение.

Синхронным резервным сервером может быть резервный сервер при физической репликации или подписчик при логической репликации. Это также может быть другой потребитель потока логической или физической репликации, способный отправлять в ответ требуемые сообщения. Помимо встроенных систем логической и физической репликации, к таким потребителям относятся специальные программы, `pg_receivewal` и `pg_recvlogical`, а также некоторые сторонние системы репликации и внешние программы. Подробнее об организации синхронной репликации с их использованием можно узнать в соответствующей документации.

25.2.8.1. Базовая настройка

При настроенной потоковой репликации установка синхронной репликации требует только дополнительной настройки: необходимо выставить synchronous_standby_names в непустое значение. Так же необходимо установить `synchronous_commit` в значение `on`, но так как это значение по умолчанию, обычно действий не требуется. (См. Подраздел 18.5.1 и Подраздел 18.6.2.) В такой конфигурации каждая транзакция будет ожидать подтверждение того, что на резервном сервере произошла запись транзакции в надёжное хранилище. Значение `synchronous_commit` может быть выставлено для отдельного пользователя, может быть прописано в файле конфигурации, для конкретного пользователя или БД или динамически изменено приложением для управления степенью надёжности на уровне отдельных транзакций.

После сохранения записи о фиксации транзакции на диске ведущего сервера эта запись WAL передаётся резервному серверу. Резервный сервер отвечает подтверждающим сообщением после сохранения каждого нового блока данных WAL на диске, если только `wal_receiver_status_interval` на нём не равен нулю. В случае, когда выбран режим `synchronous_commit remote_apply`, резервный сервер передаёт подтверждение после воспроизведения записи фиксации, когда транзакция становится видимой. Если резервный сервер выбран на роль синхронного резервного в соответствии со значением `synchronous_standby_names` на ведущем, подтверждающие сообщения с этого сервера, в совокупности с сообщениями с других синхронных серверов, будут сигналом к завершению ожидания при фиксировании транзакций, требующих подтверждения сохранения записи фиксации. Эти параметры позволяют администратору определить, какие резервные серверы будут синхронными резервными. Заметьте, что настройка синхронной репликации в основном осуществляется на главном сервере. Перечисленные в списке резервных серверы должны быть подключены к нему непосредственно; он ничего не знает о резервных серверах, подключённых каскадно, через промежуточные серверы.

Если `synchronous_commit` имеет значение `remote_write`, то в случае подтверждения транзакции ответ от резервного сервера об успешном подтверждении будет передан, когда данные запишутся в операционной системе, но не когда данные будет реально сохранены на диске. При таком значении уровень надёжности снижается по сравнению со значением `on`. Резервный сервер может потерять данные в случае падения операционной системы, но не в случае падения Postgres Pro. Тем не менее этот вариант полезен на практике, так как позволяет сократить время отклика для транзакции. Потеря данных может произойти только в случае одновременного сбоя ведущего и резервного, осложнённого повреждением БД на ведущем.

Если `synchronous_commit` имеет значение `remote_apply`, то для завершения фиксирования транзакции потребуется дождаться, чтобы текущие синхронные резервные серверы сообщили, что

они воспроизвели транзакцию и её могут видеть запросы пользователей. В простых случаях это позволяет обеспечить обычный уровень согласованности и распределение нагрузки.

Пользователи прекратят ожидание в случае запроса на быструю остановку сервера. В то время как при использовании асинхронной репликации сервер не будет полностью остановлен, пока все исходящие записи WAL не переместятся на текущий присоединённый резервный сервер.

25.2.8.2. Несколько синхронных резервных серверов

Синхронная репликация поддерживает применение одного или нескольких синхронных резервных серверов; транзакции будут ждать, пока все резервные серверы, считающиеся синхронными, не подтвердят получение своих данных. Число синхронных резервных серверов, от которых транзакции должны ждать подтверждения, задаётся в параметре `synchronous_standby_names`. В этом параметре также задаётся список имён резервных серверов и метод (`FIRST` или `ANY`) выбора синхронных из заданного списка.

С методом `FIRST` производится синхронная репликация на основе приоритетов, когда транзакции фиксируются только после того, как их записи в WAL реплицируются на заданное число синхронных резервных серверов, выбираемых согласно приоритетам. Серверы, имена которых идут в начале списка, имеют больший приоритет и выбираются на роль синхронных. Другие резервные серверы, идущие в этом списке за ними, считаются потенциальными синхронными. Если один из текущих синхронных резервных серверов по какой-либо причине отключается, он будет немедленно заменён следующим по порядку резервным сервером.

Пример значения `synchronous_standby_names` для нескольких синхронных резервных серверов, выбираемых по приоритетам:

```
synchronous_standby_names = 'FIRST 2 (s1, s2, s3)'
```

В данном примере, если работают четыре резервных сервера `s1`, `s2`, `s3` и `s4`, два сервера `s1` и `s2` будут выбраны на роль синхронных резервных, так как их имена идут в начале этого списка. Сервер `s3` будет потенциальным резервным и возьмёт на себя роль синхронного резервного при отказе `s1` или `s2`. Сервер `s4` будет асинхронным резервным, так как его имя в этом списке отсутствует.

С методом `ANY` производится синхронная репликация на основе кворума, когда транзакции фиксируются только после того, как их записи в WAL реплицируются на *как минимум* заданное число синхронных серверов в списке.

Пример значения `synchronous_standby_names` для нескольких синхронных резервных серверов, образующих кворум:

```
synchronous_standby_names = 'ANY 2 (s1, s2, s3)'
```

В данном примере, если работают четыре резервных сервера `s1`, `s2`, `s3` и `s4`, транзакции будут фиксироваться только после получения ответов как минимум от двух резервных серверов из `s1`, `s2` и `s3`. Сервер `s4` будет асинхронным резервным, так как его имя в этом списке отсутствует.

Состояние синхронности резервных серверов можно увидеть в представлении `pg_stat_replication`.

25.2.8.3. Планирование производительности

Организуя синхронную репликацию, обычно нужно обстоятельно обдумать конфигурацию и размещение резервных серверов, чтобы обеспечить приемлемую производительность приложений. Ожидание не потребляет системные ресурсы, но блокировки транзакций будут сохраняться до подтверждения передачи. Как следствие, непродуманное использование синхронной репликации приведёт к снижению производительности БД из-за увеличения времени отклика и числа конфликтов.

Postgres Pro позволяет разработчикам выбрать требуемый уровень надёжности, обеспечиваемый при репликации. Он может быть установлен для системы в целом, для отдельного пользователя или соединения или даже для отдельной транзакции.

Например, в рабочей нагрузке приложения 10% изменений могут относиться к важным данным клиентов, а 90% — к менее критичным данным, потеряв которые, бизнес вполне сможет выжить (например, это могут быть текущие разговоры пользователей между собой).

При настройке уровня синхронности репликации на уровне приложения (на ведущем) можно задать синхронную репликацию для большинства важных изменений без замедления общего рабочего ритма. Возможность настройки на уровне приложения является важным и практичным средством для получения выгод синхронной репликации при высоком быстродействии.

Следует иметь в виду, что пропускная способность сети должна быть больше скорости генерирования данных WAL.

25.2.8.4. Планирование отказоустойчивости

В `synchronous_standby_names` задаётся количество и имена синхронных резервных серверов, от которых будет ожидаться подтверждение при фиксировании транзакции, когда параметру `synchronous_commit` присвоено значение `on`, `remote_apply` или `remote_write`. Фиксирование транзакции в таком режиме может не завершиться никогда, если один из синхронных резервных серверов выйдет из строя.

Поэтому для высокой степени доступности лучше всего обеспечить наличие синхронных резервных серверов в должном количестве. Для этого можно перечислить несколько потенциальных резервных серверов в строке `synchronous_standby_names`.

При синхронной репликации на основе приоритетов синхронными резервными серверами станут серверы, имена которых стоят в этом списке первыми. Следующие за ними серверы будут становиться синхронными резервными при отказе одного из текущих.

При синхронной репликации на основе кворума кандидатами на роль синхронных резервных будут все серверы в списке. И если один из них откажет, другие серверы будут продолжать исполнять эту роль.

Когда к ведущему серверу впервые присоединяется резервный, он ещё не будет полностью синхронизированным. Это называется состоянием `навёрстывания`. Как только отставание резервного от ведущего сервера сократится до нуля в первый раз, система перейдет в состояние `потоковой передачи` в реальном времени. Сразу после создания резервного сервера навёрстывание может быть длительным. В случае выключения резервного сервера длительность этого процесса увеличится соответственно продолжительности простоя. Резервный сервер может стать синхронным только по достижении состояния `потоковой передачи`. Это состояние можно проследить в представлении `pg_stat_replication`.

Если ведущий сервер перезапускается при наличии зафиксированных транзакций, ожидающих подтверждения, эти транзакции будут помечены как полностью зафиксированные после восстановления ведущего. При этом нельзя гарантировать, что все резервные серверы успели получить все текущие данные WAL к моменту падения ведущего. Таким образом, некоторые транзакции могут считаться незафиксированными на резервном сервере, даже если они считаются зафиксированными на ведущем. Гарантия, которую мы можем дать, состоит в том, что приложение не получит явного подтверждения успешной фиксации, пока не будет уверенности, что данные WAL получены всеми синхронными резервными серверами.

Если запустить синхронные резервные серверы в указанном количестве не удаётся, вам следует уменьшить число синхронных серверов, подтверждения которых требуются для завершения фиксации транзакций, в параметре `synchronous_standby_names` (или вовсе отключить его) и перезагрузить файл конфигурации на ведущем сервере.

В случае если ведущий сервер стал недоступным для оставшихся резервных, следует переключиться на наиболее подходящий из имеющихся резервных серверов.

Если необходимо пересоздать резервный сервер при наличии ожидающей подтверждения транзакции необходимо убедиться, что команды pg_start_backup() и pg_stop_backup() запускаются

в сессии с установленным `synchronous_commit = off`, в противном случае эти запросы на подтверждение будут бесконечными для вновь возникшего резервного сервера.

25.2.9. Непрерывное архивирование на резервном сервере

Когда на резервном сервере применяется непрерывное архивирование WAL, возможны два различных сценария: архив WAL может быть общим для ведущего и резервного сервера, либо резервный сервер может иметь собственный архив WAL. Когда резервный работает с собственным архивом WAL, установите в `archive_mode` значение `always`, и он будет вызывать команду архивации для каждого сегмента WAL, который он получает при восстановлении из архива или потоковой репликации. В случае с общим архивом можно поступить аналогично, но `archive_command` должна проверять, нет ли в архиве файла, идентичного архивируемому. Таким образом, команда `archive_command` должна позаботиться о том, чтобы существующий файл не был заменён файлом с другим содержимым, а в случае попытки повторного архивирования должна сообщать об успешном выполнении. При этом все эти действия должны быть рассчитаны на условия гонки, возможные, если два сервера попытаются архивировать один и тот же файл одновременно.

Если в `archive_mode` установлено значение `on`, архивация в режиме восстановления или резерва не производится. В случае повышения резервного сервера, он начнёт архивацию после повышения, но в архив не попадут те файлы WAL или файлы истории линии времени, которые генерировал не он сам. Поэтому, чтобы в архиве оказался полный набор файлов WAL, необходимо обеспечить архивацию всех файлов WAL до того, как они попадут на резервный сервер. Это естественным образом происходит при трансляции файлов журналов, так как резервный сервер может восстановить только файлы, которые находятся в архиве, однако при потоковой репликации это не так. Когда сервер работает не в режиме резерва, различий между режимами `on` и `always` нет.

25.3. Отработка отказа

Если ведущий сервер отказывает, резервный должен начать процедуры отработки отказа.

Если отказывает резервный сервер, никакие действия по отработке отказа не требуются. Если резервный сервер будет перезапущен, даже через некоторое время, немедленно начнётся операция восстановления, благодаря возможности возобновляемого восстановления. Если вернуть резервный сервер в строй невозможно, необходимо создать полностью новый экземпляр резервного сервера.

Когда ведущий сервер отказывает и резервный сервер становится новым ведущим, а затем старый ведущий включается снова, необходим механизм для предотвращения возврата старого к роли ведущего. Иногда его называют STONITH (Shoot The Other Node In The Head, «Выстрелите в голову другому узлу»), что позволяет избежать ситуации, когда обе системы считают себя ведущими, и в результате возникают конфликты и потеря данных.

Во многих отказоустойчивых конструкциях используются всего две системы: ведущая и резервная, с некоторым контрольным механизмом, который постоянно проверяет соединение между ними и работоспособность ведущей. Также возможно применение третьей системы (называемой следящим сервером) для исключения некоторых вариантов нежелательной отработки отказа, но эта дополнительная сложность оправдана, только если вся схема достаточно хорошо продумана и тщательно протестирована.

Postgres Pro не предоставляет системного программного обеспечения, необходимого для определения сбоя на ведущем и уведомления резервного сервера баз данных. Имеется множество подобных инструментов, которые хорошо интегрируются со средствами операционной системы, требуемыми для успешной отработки отказа, например, для миграции IP-адреса.

Когда происходит переключение на резервный сервер, только один сервер продолжает работу. Это состояние называется ущербным. Бывший резервный сервер теперь является ведущим, а бывший ведущий отключён и может оставаться отключённым. Для возвращения к нормальному состоянию необходимо запустить новый резервный сервер, либо на бывшем ведущем, либо в третьей, возможно, новой системе. Ускорить этот процесс в больших кластерах позволяет утилита

pg_rewind. По завершении этого процесса можно считать, что ведущий и резервный сервер поменялись ролями. Некоторые используют третий сервер в качестве запасного для нового ведущего, пока не будет воссоздан новый резервный сервер, хотя это, очевидно, усложняет конфигурацию системы и рабочие процедуры.

Таким образом, переключение с ведущего сервера на резервный может быть быстрым, но требует некоторого времени для повторной подготовки отказоустойчивого кластера. Регулярные переключения с ведущего сервера на резервный полезны, так как при этом появляется плановое время для отключения и проведения обслуживания. Это также позволяет убедиться в работоспособности механизма отработки отказа и гарантировать, что он действительно будет работать, когда потребуется. Эти административные процедуры рекомендуется документировать письменно.

Чтобы сделать ведущим резервный сервер, принимающий журналы, выполните команду `pg_ctl promote` или создайте файл-триггер с именем и путём, заданным в параметре `trigger_file` в файле `recovery.conf`. Если для переключения планируется использовать команду `pg_ctl promote`, указывать `trigger_file` не требуется. Если резервный сервер применяется для анализа данных, чтобы только разгрузить ведущий, выполняя запросы на чтение, а не обеспечивать отказоустойчивость, повышать его до ведущего не понадобится.

25.4. Другие методы трансляции журнала

Встроенному режиму резерва, описанному в предыдущем разделе, есть альтернатива — задать в `restore_command` команду, следящую за содержимым архива. В версии 8.4 и ниже был возможен только такой вариант. В такой конфигурации режим `standby_mode` выключается, так как реализуется отдельный механизм слежения за данными, требующихся для резервного сервера. См. модуль pg_standby для примера реализации такой возможности.

Необходимо отметить, что в этом режиме сервер будет применять только один файл WAL одновременно, то есть если использовать резервный сервер для запросов (см. сервер горячего резерва), будет задержка между операциями на главном и моментом видимости этой операции резервным, соответствующей времени заполнения файла WAL. `archive_timeout` можно использовать для снижения этой задержки. Так же необходимо отметить, что нельзя совмещать этот метод с потоковой репликацией.

В процессе работы на ведущем сервере и резервном будет происходить обычное формирование архивов и их восстановление. Единственной точкой соприкосновения двух серверов будут только архивы файлов WAL на обеих сторонах: на ведущем архивы формируются, на резервном происходит чтение данных из архивов. Следует внимательно следить за тем, чтобы архивы WAL от разных ведущих серверов не смешивались или не перепутывались. Архив не должен быть больше, чем это необходимо для работы резерва.

Магия, заставляющая работать вместе два слабо связанных сервера, проста: `restore_command`, выполняющаяся на резервном при запросе следующего файла WAL, ожидает его доступности на ведущем. Команда `restore_command` задаётся в файле `recovery.conf` на резервном сервере. Обычно процесс восстановления запрашивает файл из архива WAL, сообщая об ошибке в случае его недоступности. В работе резервного сервера недоступность очередного файла WAL является обычной ситуацией, поэтому резервный просто ожидает его появления. Для файлов, оканчивающихся на `.history`, ожидание не требуется, поэтому возвращается ненулевой код. Ожидающая `restore_command` может быть написана как пользовательский скрипт, который в цикле опрашивает, не появился ли очередной файл WAL. Также должен быть способ инициировать переключение роли, при котором цикл в `restore_command` должен прерваться, а резервный сервер должен получить ошибку «файл не найден». При этом восстановление завершится, и резервный сервер сможет стать обычным.

Псевдокод для подходящей `restore_command`:

```
triggered = false;
while (!NextWALFileReady() && !triggered)
```

```
{
    sleep(100000L);          /* ждать ~0.1 сек*/
    if (CheckForExternalTrigger())
        triggered = true;
}
if (!triggered)
        CopyWALFileForRecovery();
```

Рабочий пример ожидающей `restore_command` представлен в модуле pg_standby. К нему следует обратится за примером правильной реализации логики, описанной выше. Он так же может быть расширен для поддержки особых конфигураций и окружений.

Метод вызова переключения является важной частью планирования и архитектуры. Один из возможных вариантов — команда `restore_command`. Она исполняется единожды для каждого файла WAL, но процесс, запускаемый `restore_command`, создаётся и завершается для каждого файла, так что это не служба и не серверный процесс, и применить сигналы и реализовать их обработчик в нём нельзя. Поэтому `restore_command` не подходит для отработки отказа. Можно организовать переключение по тайм-ауту, в частности, связав его с известным значением `archive_timeout` на ведущем. Однако это не очень надёжно, так как переключение может произойти и из-за проблем в сети или загруженности ведущего сервера. В идеале для этого следует использовать механизм уведомлений, например явно создавать файл-триггер, если это возможно.

25.4.1. Реализация

Сокращённая процедура настройки для резервного сервера с применением альтернативного метода указана ниже. Для подробностей по каждому шагу следует обратиться к указанному разделу.

1. Разверните ведущую и резервную системы, сделав их максимально одинаковыми, включая две одинаковые копии Postgres Pro одного выпуска.
2. Настройте постоянную архивацию с ведущего сервера в каталог архивов WAL на резервном. Убедитесь, что archive_mode, archive_command и archive_timeout установлены в соответствующие значения на ведущем (см. Подраздел 24.3.1).
3. Создайте базовую копию данных ведущего сервера (см. Подраздел 24.3.2) и восстановите её на резервном.
4. Запустите восстановление на резервном сервере из локального архива WAL с помощью команды `restore_command` из файла `recovery.conf`, как описано выше (см. Подраздел 24.3.4).

Поток восстановления только читает архив WAL, поэтому, как только файл WAL скопирован на резервную систему, его можно копировать на ленту в то время, как его читает резервный сервер. Таким образом, работа резервного сервера в целях отказоустойчивости может быть совмещена с долговременным сохранением файлов для восстановления после катастрофических сбоев.

Для целей тестирования возможен запуск ведущего и резервного сервера в одной системе. Это не обеспечивает надёжность серверов, так же как и не подходит под описание высокой доступности.

25.4.2. Построчная трансляция журнала

Так же возможна реализация построчной трансляции журналов с применением альтернативного метода, хотя это требует дополнительных доработок, а изменения будут видны для запросов на сервере горячего резерва только после передачи полного файла WAL.

Внешняя программа может вызвать функцию `pg_walfile_name_offset()` (см. Раздел 9.26) для поиска имени файла и точного смещения в нём от текущего конца WAL. Можно получить доступ к файлу WAL напрямую и скопировать данные из последнего известного окончания WAL до текущего окончания на резервном сервере. При таком подходе интервал возможной потери данных определяется временем цикла работы программы копирования, что может составлять очень малую величину. Так же не потребуется напрасно использовать широкую полосу пропускания

для принудительного архивирования частично заполненного файла сегмента. Следует отметить, что на резервном сервере скрипт команды `restore_command` работает только с файлом WAL целиком, таким образом, копирование данных нарастающим итогом не может быть выполнено на резервном обычными средствами. Это используется только в случае отказа ведущего — когда последний частично сформированный файл WAL предоставляется резервному непосредственно перед переключением. Корректная реализация этого процесса требует взаимодействия скрипта команды `restore_command` с данными из программы копирования.

Начиная с PostgreSQL версии 9.0 можно использовать потоковую репликацию (см. Подраздел 25.2.5) для получения этих же преимуществ меньшими усилиями.

25.5. Горячий резерв

Термин «горячий резерв» используется для описания возможности подключаться к серверу и выполнять запросы на чтение, в то время как сервер находится в режиме резерва или восстановления архива. Это полезно и для целей репликации, и для восстановления желаемого состояния из резервной копии с высокой точностью. Так же термин «горячий резерв» описывает способность сервера переходить из режима восстановления к обычной работе, в то время как пользователи продолжают выполнять запросы и/или их соединения остаются открытыми.

В режиме горячего резерва запросы выполняются примерно так же, как и в обычном режиме, с некоторыми отличиями в использовании и администрировании, описанными ниже.

25.5.1. Обзор на уровне пользователя

Когда параметр hot_standby на резервном сервере установлен в true, то он начинает принимать соединения сразу как только система придёт в согласованное состояние в процессе восстановления. Для таких соединений будет разрешено только чтение, запись невозможна даже во временные таблицы.

Для того, чтобы данные с ведущего сервера были получены на резервном, требуется некоторое время. Таким образом, имеется измеряемая задержка между ведущим и резервным серверами. Поэтому запуск одинаковых запросов примерно в одно время на ведущем и резервном серверах может вернуть разный результат. Можно сказать, что данные на резервном сервере *в конечном счёте согласуются* с ведущим. После того как запись о зафиксированной транзакции воспроизводится на резервном сервере, изменения, совершённые в этой транзакции, становится видны в любых последующих снимках данных на резервном сервере. Снимок может быть сделан в начале каждого запроса или в начале каждой транзакции в зависимости от уровня изоляции транзакции. Более подробно см. Раздел 13.2.

Транзакции, запущенные в режиме горячего резерва, могут выполнять следующие команды:

- Доступ к данным — `SELECT`, `COPY TO`
- Команды для работы с курсором — `DECLARE`, `FETCH`, `CLOSE`
- Параметры — `SHOW`, `SET`, `RESET`
- Команды явного управления транзакциями
  - `BEGIN`, `END`, `ABORT`, `START TRANSACTION`
  - `SAVEPOINT`, `RELEASE`, `ROLLBACK TO SAVEPOINT`
  - Блок `EXCEPTION` и другие внутренние подчиненные транзакции
- `LOCK TABLE`, только когда исполняется в явном виде в следующем режиме: `ACCESS SHARE`, `ROW SHARE` или `ROW EXCLUSIVE`.
- Планы и ресурсы — `PREPARE`, `EXECUTE`, `DEALLOCATE`, `DISCARD`
- Дополнения и расширения — `LOAD`
- `UNLISTEN`

Транзакции, запущенные в режиме горячего резерва, никогда не получают ID транзакции и не могут быть записаны в журнал предзаписи. Поэтому при попытке выполнить следующие действия возникнут ошибки:

- Команды манипуляции данными (DML) — `INSERT`, `UPDATE`, `DELETE`, `COPY FROM`, `TRUNCATE`. Следует отметить, что нет разрешённых действий, которые приводили бы к срабатыванию триггера во время исполнения на резервном сервере. Это ограничение так же касается и временных таблиц, так как строки таблицы не могут быть прочитаны или записаны без обращения к ID транзакции, что в настоящее время не возможно в среде горячего резерва.
- Команды определения данных (DDL) — `CREATE`, `DROP`, `ALTER`, `COMMENT`. Эти ограничения так же относятся и к временным таблицам, так как операции могут потребовать обновления таблиц системных каталогов.
- `SELECT ... FOR SHARE | UPDATE`, так как блокировка строки не может быть проведена без обновления соответствующих файлов данных.
- Правила для выражений `SELECT`, которые приводят к выполнению команд DML.
- `LOCK` которая явно требует режим более строгий чем `ROW EXCLUSIVE MODE`.
- `LOCK` в короткой форме с умолчаниями, так как требует `ACCESS EXCLUSIVE MODE`.
- Команды управления транзакциями, которые в явном виде требуют режим не только для чтения
  - `BEGIN READ WRITE`, `START TRANSACTION READ WRITE`
  - `SET TRANSACTION READ WRITE`, `SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE`
  - `SET transaction_read_only = off`
- Команды двухфазной фиксации — `PREPARE TRANSACTION`, `COMMIT PREPARED`, `ROLLBACK PREPARED`, так как даже транзакции только для чтения нуждаются в записи в WAL на подготовительной фазе (первая фаза двухфазной фиксации).
- Обновление последовательностей — `nextval()`, `setval()`
- `LISTEN`, `NOTIFY`

При обычной работе транзакции «только для чтения» могут использовать команды `LISTEN` и `NOTIFY`; таким образом, сеансы горячего резерва работают с несколько большими ограничениями, чем обычные только читающие сеансы. Возможно, что некоторые из этих ограничений будут ослаблены в следующих выпусках.

В режиме горячего резерва параметр `transaction_read_only` всегда имеет значение true и изменить его нельзя. Но если не пытаться модифицировать содержимое БД, подключение к серверу в этом режиме не отличается от подключений к обычным базам данных. При отработке отказа или переключении ролей база данных переходит в обычный режим работы. Когда сервер меняет режим работы, установленные сеансы остаются подключёнными. После выхода из режима горячего резерва становится возможным запускать пишущие транзакции (даже в сеансах, начатых ещё в режиме горячего резерва).

Пользователи могут узнать, активен ли режим горячего резерва в их сеансе, выполнив команду `SHOW transaction_read_only`. Кроме того, пользователи могут получить информацию о резервном сервере, используя ряд функций (см. Таблицу 9.80). Эти средства позволяют как создавать программы, учитывающие текущий статус базы данных, так и отслеживать процесс восстановления или разрабатывать более сложные программы для восстановления баз данных в нужном состоянии.

25.5.2. Обработка конфликтов запросов

Ведущий и резервный серверы связаны между собой многими слабыми связями. События на ведущем сервере оказывают влияние на резервный. В результате имеется потенциальная возможность отрицательного влияния или конфликта между ними. Наиболее простой для

понимания конфликт — быстродействие: если на ведущем происходит загрузка очень большого объёма данных, то происходит создание соответствующего потока записей WAL на резервный сервер. Таким образом, запросы на резервном конкурируют за системные ресурсы, например, ввод-вывод.

Так же может возникнуть дополнительный тип конфликта на сервере горячего резерва. Этот конфликт называется *жёстким конфликтом*, оказывает влияние на запросы, приводя к их отмене, а в некоторых случаях и к обрыву сессии для разрешения конфликтов. Пользователям предоставлен набор средств для обработки подобных конфликтов. Случаи конфликтов включают:

- Установка эксклюзивной блокировки на ведущем сервере, как с помощью явной команды `LOCK`, так и при различных DDL, что приводит к конфликту доступа к таблицам на резервном.
- Удаление табличного пространства на ведущем сервере приводит к конфликту на резервном когда запросы используют это пространство для хранения временных рабочих файлов.
- Удаление базы данных на ведущем сервере конфликтует с сессиями, подключёнными к этой БД на резервном.
- Приложение очистки устаревших транзакций из WAL конфликтует с транзакциями на резервном сервере, которые используют снимок данных, который всё ещё видит какие-то из очищенных на ведущем строк.
- Приложение очистки устаревших транзакций из WAL конфликтует с запросами к целевой странице на резервном сервере вне зависимости от того, являются ли данные удалёнными или видимыми.

В этих случаях на ведущем сервере просто происходит ожидание; пользователю следует выбрать какую их конфликтующих сторон отменить. Тем не менее на резервном нет выбора: действия из WAL уже произошли на ведущем, поэтому резервный обязан применить их. Более того, позволять обработчику WAL ожидать неограниченно долго может быть крайне нежелательно, так как отставание резервного сервера от ведущего может всё возрастать. Таким образом, механизм обеспечивает принудительную отмену запросов на резервном сервере, которые конфликтуют с применяемыми записями WAL.

Примером такой проблемы может быть ситуация: администратор на ведущем сервере выполнил команду `DROP TABLE` для таблицы, которая сейчас участвует в запросе на резервном. Понятно, что этот запрос нельзя будет выполнять дальше, если команда `DROP TABLE` применится на резервном. Если бы этот запрос выполнялся на ведущем, команда `DROP TABLE` ждала бы его окончания. Но когда на ведущем выполняется только команда `DROP TABLE`, ведущий сервер не знает, какие запросы выполняются на резервном, поэтому он не может ждать завершения подобных запросов. Поэтому если записи WAL с изменением прибудут на резервный сервер, когда запрос будет продолжать выполняться, возникнет конфликт. В этом случае резервный сервер должен либо задержать применение этих записей WAL (и всех остальных, следующих за ними), либо отменить конфликтующий запрос, чтобы можно было применить `DROP TABLE`.

Если конфликтный запрос короткий, обычно желательно разрешить ему завершиться, ненадолго задержав применение записей WAL, но слишком большая задержка в применении WAL обычно нежелательна. Поэтому механизм отмены имеет параметры max_standby_archive_delay и max_standby_streaming_delay, которые определяют максимально допустимое время задержки применения WAL. Конфликтующие запросы будут отменены, если они длятся дольше допустимого времени задержки применения очередных записей WAL. Два параметра существуют для того, чтобы можно было задать разные значения для чтения записей WAL из архива (то есть при начальном восстановлении из базовой копии либо при «навёрстывании» ведущего сервера в случае большого отставания) и для получения записей WAL при потоковой репликации.

На резервном сервере, созданном преимущественно для отказоустойчивости, лучше выставлять параметры задержек относительно небольшими, чтобы он не мог сильно отстать от ведущего из-за задержек, связанных с ожиданием запросов горячего резерва. Однако если резервный сервер предназначен для выполнения длительных запросов, то высокое значение или даже бесконечное ожидание могут быть предпочтительнее. Тем не менее следует иметь в виду, что длительные

запросы могут оказать влияние на другие сессии на резервном сервере в виде отсутствия последних изменений от ведущего из-за задержки применения записей WAL.

В случае, если задержка, определённая `max_standby_archive_delay` или `max_standby_streaming_delay` будет превышена, конфликтующий запрос будет отменён. Обычно это выражается в виде ошибки отмены, но в случае проигрывания команды `DROP DATABASE` обрывается вся конфликтная сессия. Так же, если конфликт произошел при блокировке, вызванной транзакцией в состоянии IDLE, конфликтная сессия разрывается (это поведение может изменить в будущем).

Отменённые запросы могут быть немедленно повторены (конечно после старта новой транзакции). Так как причина отмены зависит от природы проигрываемых записей WAL, запрос, который был отменён, может быть успешно выполнен вновь.

Следует учесть, что параметры задержки отсчитываются от времени получения резервным сервером данных WAL. Таким образом, период дозволенной работы для запроса на резервном сервере никогда не может быть длиннее параметра задержки и может быть существенно короче, если резервный уже находится в режиме задержки в результате ожидания предыдущего запроса или результат не доступен из-за высокой нагрузки обновлений.

Наиболее частой причиной конфликтов между запросами на резервном сервере и проигрыванием WAL является преждевременная очистка. Обычно Postgres Pro допускает очистку старых версий записей при условии что ни одна из транзакций их не видит согласно правилам видимости данных для MVCC. Тем не менее эти правила применяются только для транзакций, выполняемых на главном сервере. Таким образом, допустима ситуация, когда на главном запись уже очищена, но эта же запись всё ещё видна для транзакций на резервном сервере.

Для опытных пользователей следует отметить, что как очистка старых версий строк, так и заморозка версии строки могут потенциально вызвать конфликт с запросами на резервном сервере. Ручной запуск команды `VACUUM FREEZE` может привести к конфликту, даже в таблице без обновленных и удалённых строк.

Пользователи должны понимать, что регулярное и активное изменение данных в таблицах на ведущем сервере чревато отменой длительных запросов на резервном. В таком случае установка конечного значения для `max_standby_archive_delay` или `max_standby_streaming_delay` действует подобно ограничению `statement_timeout`.

В случае, если количество отменённых запросов на резервном сервере получается неприемлемым, существует ряд дополнительных возможностей. Первая возможность — установить параметр `hot_standby_feedback`, который не даёт команде `VACUUM` удалять записи, ставшие недействительными недавно, что предотвращает конфликты очистки. При этом следует учесть, что это вызывает задержку очистки мёртвых строк на ведущем, что может привести к нежелательному распуханию таблицы. Тем не менее в итоге ситуация будет не хуже, чем если бы запросы к резервному серверу исполнялись непосредственно на ведущем, но при этом сохранится положительный эффект от разделения нагрузки. В случае, когда соединение резервных серверов с ведущим часто разрывается, следует скорректировать период, в течение которого обратная связь через `hot_standby_feedback` не обеспечивается. Например, следует подумать об увеличении `max_standby_archive_delay`, чтобы запросы отменялись не сразу при конфликтах с архивом WAL в период разъединения. Также может иметь смысл увеличить `max_standby_streaming_delay` для предотвращения быстрой отмены запросов из-за полученных записей WAL после восстановления соединения.

Другая возможность — увеличение vacuum_defer_cleanup_age на ведущем сервере таким образом, чтобы мёртвые записи не очищались бы так быстро, как при обычном режиме работы. Это даёт запросам на резервном сервере больше времени на выполнение, прежде чем они могут быть отменены, без увеличения задержки `max_standby_streaming_delay`. Тем не менее при таком подходе очень трудно обеспечить какое-то определённое окно по времени, так как `vacuum_defer_cleanup_age` измеряется в количестве транзакций, выполняемых на ведущем сервере.

Количество отменённых запросов и причины отмены можно просмотреть через системное представление `pg_stat_database_conflicts` на резервном сервере. Системное представление `pg_stat_database` так же содержит итоговую информацию.

25.5.3. Обзор административной части

Если в файле `postgresql.conf` для параметра `hot_standby` задано значение `on` (по умолчанию) и существует файл `recovery.conf`, сервер запустится в режиме горячего резерва. Однако может пройти некоторое время, прежде чем к нему можно будет подключиться, так как он не будет принимать подключения, пока не произведёт восстановление до согласованного состояния, подходящего для выполнения запросов. (Информация о согласованности состояния записывается на ведущем сервере в контрольной точке.) В течение этого периода клиенты при попытке подключения будут получать сообщение об ошибке. Убедиться, что сервер включился в работу, можно либо повторяя попытки подключения из приложения до успешного подключения, либо дождавшись появления в журналах сервера таких сообщений:

```
LOG:  entering standby mode

... then some time later ...

LOG:  consistent recovery state reached
LOG:  database system is ready to accept read only connections
```

Включить горячий резерв нельзя, если WAL был записан в период, когда на ведущем сервере параметр `wal_level` имел значение, отличное от `replica` и `logical`. Достижение согласованного состояния также может быть отсрочено, если имеют место оба этих условия:

- Пишущая транзакция имеет более 64 подтранзакций
- Очень длительные пишущие транзакции

Если вы применяете файловую репликацию журналов («тёплый резерв»), возможно, придётся ожидать прибытия следующего файла WAL (максимальное время ожидания задаётся параметром `archive_timeout` на ведущем сервере).

Значения некоторых параметров на резервном сервере необходимо изменить при модификации их на ведущем. Для таких параметров значения на резервном сервере должны быть равны или больше значений на ведущем. Если параметр имеет недостаточно большое значение, резервный сервер не сможет начать работу. Следует увеличить значение и повторить попытку восстановления ещё раз. Это касается следующих параметров:

- `max_connections`
- `max_prepared_transactions`
- `max_locks_per_transaction`
- `max_worker_processes`

Очень важно для администратора выбрать подходящие значения для max_standby_archive_delay и max_standby_streaming_delay. Оптимальное значение зависит от приоритетов. Например, если основное назначение сервера — обеспечение высокой степени доступности, то следует установить короткий период, возможно даже нулевой, хотя это очень жёсткий вариант. Если резервный сервер планируется как дополнительный сервер для аналитических запросов, то приемлемой будет максимальная задержка в несколько часов или даже -1, что означает бесконечное ожидание окончания запроса.

Вспомогательные биты статуса транзакций, записанные на ведущем, не попадают в WAL, так что они, скорее всего, будут перезаписаны на нём при работе с данными. Таким образом, резервный сервер будет производить запись на диск, даже если все пользователи только читают данные, ничего не меняя. Кроме того, пользователи будут записывать временные файлы при сортировке больших объёмов и обновлять файлы кеша. Поэтому в режиме горячего резерва ни одна часть базы данных фактически не работает в режиме «только чтение». Следует отметить, что также возможно

выполнить запись в удалённую базу данных с помощью модуля dblink и другие операции вне базы данных с применением PL-функций, несмотря на то, что транзакции по-прежнему смогут только читать данные.

Следующие типы административных команд недоступны в течение режима восстановления:

- Команды определения данных (DDL) — например, `CREATE INDEX`
- Команды выдачи привилегий и назначения владельца — `GRANT`, `REVOKE`, `REASSIGN`
- Команды обслуживания — `ANALYZE`, `VACUUM`, `CLUSTER`, `REINDEX`

Ещё раз следует отметить, что некоторые из этих команд фактически доступны на ведущем сервере для транзакций в режиме только для чтения.

В результате нельзя создать дополнительные индексы или статистику, чтобы они существовали только на резервном. Если подобные административные команды нужны, то их следует выполнить на ведущем сервере, затем эти изменения будут распространены на резервные серверы.

Функции `pg_cancel_backend()` и `pg_terminate_backend()` работают на стороне пользователя, но не для процесса запуска, который обеспечивает восстановление. Представление `pg_stat_activity` не показывает восстанавливаемые транзакции как активные. Поэтому представление `pg_prepared_xacts` всегда пусто в ходе восстановления. Если требуется разобрать сомнительные подготовленные транзакции, следует обратиться к `pg_prepared_xacts` на ведущем и выполнить команды для разбора транзакций там либо разобрать их по окончании восстановления.

`pg_locks` отображает блокировки, происходящие в процессе работы сервера как обычно. `pg_locks` так же показывает виртуальные транзакции, обработанные процессом запуска, которому принадлежат все `AccessExclusiveLocks`, наложенные транзакциями в режиме восстановления. Следует отметить, что процесс запуска не запрашивает блокировки, чтобы внести изменения в базу данных, поэтому блокировки, отличные от `AccessExclusiveLocks` не показываются в `pg_locks` для процесса запуска, подразумевается их существование.

Модуль check_pgsql для Nagios будет работать, так как сервер выдаёт простую информацию, наличие которой он проверяет. Скрипт мониторинга check_postgres так же работает, хотя для некоторых выдаваемых показателей результаты могут различаться или вводить в заблуждение. Например, нельзя отследить время последней очистки, так как очистка не производится на резервном сервере. Очистка запускается на ведущем сервере и результаты её работы передаются резервному.

Команды управления файлами WAL, например `pg_start_backup`, `pg_switch_wal` и т. д. не будут работать во время восстановления.

Динамически загружаемые модули работать будут, включая `pg_stat_statements`.

Рекомендательная блокировка работает обычно при восстановлении, включая обнаружение взаимных блокировок. Следует отметить, что рекомендательная блокировка никогда не попадает в WAL, таким образом для рекомендательной блокировки как на ведущем сервере, так и на резервном, невозможен конфликт с проигрыванием WAL. Но возможно получение рекомендательной блокировки на ведущем сервере, а затем получение подобной рекомендательной блокировки на резервном. Рекомендательная блокировка относится только к серверу, на котором она получена.

Системы репликации на базе триггеров, подобные Slony, Londiste и Bucardo не могут запускаться на резервном сервере вовсе, хотя они превосходно работают на ведущем до тех пор, пока не будет подана команда не пересылать изменения на резервный. Проигрывание WAL не основано на триггерах, поэтому поток WAL нельзя транслировать с резервного сервера в другую систему, которая требует дополнительной записи в БД или работает на основе триггеров.

Новые OID не могут быть выданы, хотя, например генераторы UUID смогут работать, если они не пытаются записывать новое состояние в базу данных.

В настоящий момент создание временных таблиц недопустимо при транзакции только для чтения, в некоторых случаях существующий скрипт будет работать неверно. Это ограничение может быть ослаблено в следующих выпусках. Это одновременно требование SQL стандарта и техническое требование.

Команда `DROP TABLESPACE` может быть выполнена только если табличное пространство пусто. Некоторые пользователи резервного сервера могут активно использовать табличное пространство через параметр `temp_tablespaces`. Если имеются временные файлы в табличных пространствах, все активные запросы отменяются для обеспечения удаления временных файлов, затем табличное пространство может быть удалено и продолжено проигрывание WAL.

Выполнение команды `DROP DATABASE` или `ALTER DATABASE ... SET TABLESPACE` на ведущем сервере приводит к созданию записи в WAL, которая вызывает принудительное отключение всех пользователей, подключённых к этой базе данных на резервном. Это происходит немедленно, вне зависимости от значения `max_standby_streaming_delay`. Следует отметить, что команда `ALTER DATABASE ... RENAME` не приводит к отключению пользователей, так что обычно она действует незаметно, хотя в некоторых случаях возможны сбои программ, которые зависят от имени базы данных.

Если вы в обычном режиме (не в режиме восстановления) выполните `DROP USER` или `DROP ROLE` для роли с возможностью подключения, в момент, когда этот пользователь подключён, на данном пользователе это никак не отразится — он останется подключённым. Однако переподключиться он уже не сможет. Это же поведение действует в режиме восстановления — если выполнить `DROP USER` на ведущем сервере, пользователь не будет отключён от резервного.

Сборщик статистики работает во время восстановления. Все операции сканирования, чтения, блоки, использование индексов и т. п. будут записаны обычным образом на резервном сервере. Действия, происходящие при проигрывании, не будут дублировать действия на ведущем сервере, то есть проигрывание команды вставки не увеличит значение столбца Inserts в представлении pg_stat_user_tables. Файлы статистики удаляются с началом восстановления, таким образом, статистика на ведущем сервере и резервном будет разной. Это является особенностью, не ошибкой.

Автоматическая очистка не работает во время восстановления. Она запустится в обычном режиме после завершения восстановления.

Во время восстановления работает процесс контрольных точек и процесс фоновой записи. Процесс контрольных точек обрабатывает точки перезапуска (подобные контрольным точкам на ведущем сервере), а процесс фоновой записи выполняет обычные операции по очистке блоков. В том числе он может обновлять вспомогательные биты, сохранённые на резервном сервере. Во время восстановления принимается команда `CHECKPOINT`, но она производит точку перезапуска, а не создаёт новую точку восстановления.

25.5.4. Ссылки на параметры горячего резерва

Различные параметры были упомянуты выше в Подразделе 25.5.2 и Подразделе 25.5.3.

На ведущем могут применяться параметры wal_level и vacuum_defer_cleanup_age. Параметры max_standby_archive_delay и max_standby_streaming_delay на ведущем не действуют.

На резервном сервере могут применяться параметры hot_standby, max_standby_archive_delay и max_standby_streaming_delay. Параметр vacuum_defer_cleanup_age на нём не действует, пока сервер остаётся в режиме резервного сервера. Но если он станет ведущим, его значение вступит в силу.

25.5.5. Ограничения

Имеются следующие ограничения горячего резерва. Они могут и скорее всего будут исправлены в следующих выпусках:

- Требуется информация о всех запущенных транзакциях перед тем как будет создан снимок данных. Транзакции, использующие большое количество подтранзакций (в настоящий момент больше 64), будут задерживать начало соединения только для чтения до завершения самой длинной пишущей транзакции. При возникновении этой ситуации поясняющее сообщение будет записано в журнал сервера.
- Подходящие стартовые точки для запросов на резервном сервере создаются при каждой контрольной точке на главном. Если резервный сервер отключается, в то время как главный был в отключённом состоянии, может оказаться невозможным возобновить его работу в режиме горячего резерва, до того, как запустится ведущий и добавит следующие стартовые точки в журналы WAL. Подобная ситуация не является проблемой для большинства случаев, в которых она может произойти. Обычно, если ведущий сервер выключен и больше не доступен, это является следствием серьёзного сбоя и в любом случае требует преобразования резервного в новый ведущий. Так же в ситуации, когда ведущий отключён намеренно, проверка готовности резервного к преобразованию в ведущий тоже является обычной процедурой.
- В конце восстановления блокировки `AccessExclusiveLocks`, вызванные подготовленными транзакциями, требуют удвоенное, в сравнении с нормальным, количество блокировок записей таблицы. Если планируется использовать либо большое количество конкурирующих подготовленных транзакций, обычно вызывающие `AccessExclusiveLocks`, либо большие транзакции с применением большого количества `AccessExclusiveLocks`, то рекомендуется выбрать большое значение параметра `max_locks_per_transaction`, возможно в два раза большее, чем значение параметра на ведущем сервере. Всё это не имеет значения, когда `max_prepared_transactions` равно 0.
- Уровень изоляции транзакции Serializable в настоящее время недоступен в горячем резерве. (За подробностями обратитесь к Подразделу 13.2.3 и Подразделу 13.4.1) Попытка выставить для транзакции такой уровень изоляции в режиме горячего резерва вызовет ошибку.

Глава 26. Конфигурация восстановления

Глава описывает параметры, задаваемые в файле `recovery.conf`. Они применяются лишь во время восстановления и должны заново устанавливаться перед каждым последующим восстановлением. Содержимое файла перезагружается при получении основным серверным процессом сигнала `SIGHUP`.

Параметры в файле `recovery.conf` указываются в формате `name = 'value'`. Каждый параметр должен располагаться на отдельной строке. Для строчного комментария используется хеш (`#`). Чтобы включить апостроф в значение параметра, продублируйте его (`''`).

Пример файла `share/recovery.conf.sample`, поставляемый в рамках дистрибутива, размещён в каталоге `share/`.

26.1. Параметры восстановления из архива

`restore_command` (`string`)

Команда оболочки ОС, которая выполняется для извлечения архивного сегмента файлов WAL. Этот параметр требуется для восстановления из архива, но необязателен для потоковой репликации. Любое вхождение `%f` в строке заменяется именем извлекаемого из архива файла, а `%p` заменяется на путь назначения при копировании на сервере. (Путь указывается относительно текущего рабочего каталога, т. е. относительно каталога хранения данных кластера.) Любое вхождение `%r` заменяется на имя файла, в котором содержится последняя действительная точка восстановления. Это самый ранний файл, который необходимо хранить для возможности восстановления, и эту информацию можно использовать для усечения архива в целях его минимизации. `%r` обычно используется при организации тёплого резерва (см. Раздел 25.2). Для того чтобы указать символ `%`, продублируйте его (`%%`).

Обратите внимание, что команда должна возвращать ноль на выходе лишь в случае успешного выполнения. У команды *будет* запрошен список имён файлов, которые отсутствуют в архиве; в этом случае она должна возвращать ненулевой статус. Примеры:

```
restore_command = 'cp /mnt/server/archivedir/%f "%p"'
restore_command = 'copy "C:\\server\\archivedir\\%f" "%p"'  # Windows
```

В случае прерывания команды сигналом (отличным от SIGTERM, который используется для остановки сервера баз данных) или при возникновении ошибки оболочки (например, если команда не найдена), процесс восстановления будет остановлен и сервер не запустится.

`archive_cleanup_command` (`string`)

Этот необязательный параметр указывает команду оболочки ОС, которая будет вызываться при каждой точке перезапуска. Назначение команды `archive_cleanup_command` в том, чтобы предоставить механизм очистки от старых архивных файлов WAL, которые более не нужны на резервном сервере. Любое вхождение `%r` заменяется на имя файла, содержащего последнюю действительную точку восстановления. Это самый ранний файл, который необходимо *хранить* для возможности восстановления, а более старые файлы, чем `%r`, могут быть безболезненно удалены. Эта информация может быть использована для усечения архива с целью его минимизации при сохранении возможности последующего восстановления из заданной точки. Модуль pg_archivecleanup часто используется в качестве значения параметра `archive_cleanup_command` в конфигурациях с одним резервным сервером, например:

```
archive_cleanup_command = 'pg_archivecleanup /mnt/server/archivedir %r'
```

Стоит обратить внимание, что в конфигурациях с множеством резервных серверов, использующих общий архивный каталог для восстановления, необходимо контролировать удаление файлов WAL, так как для некоторых они ещё могут требоваться. Поэтому `archive_cleanup_command` обычно используется при организации тёплого резерва (см. Раздел 25.2). Для того чтобы указать символ `%` в команде, он пишется дважды `%%`.

В случаях, когда команда возвращает ненулевой статус завершения, в журнал записывается предупреждающее сообщение. Если же команда прерывается сигналом или оболочка ОС выдаёт ошибку (например, команда не найдена), вызывается фатальная ошибка.

`recovery_end_command` (`string`)

Этот параметр задаёт команду оболочки, которая будет выполнена единожды в конце процесса восстановления. Назначение параметра `recovery_end_command` — предоставить механизм для очистки после репликации или восстановления. Любое вхождение `%r` заменяется именем файла, содержащим последнюю действительную точку восстановления, например, как в archive_cleanup_command.

Если команда вернёт ненулевой статус завершения, то в журнал будет записано предупреждающее сообщение, но при этом запуск сервера будет продолжен. За исключением случаев, когда команда прервана сигналом или при возникновения ошибки оболочки ОС (например, команда не найдена). В таких случаях база данных не будет запускаться.

26.2. Параметры управления восстановлением

По умолчанию процесс восстановления производится вплоть до окончания журнала WAL. Нижеуказанные параметры могут использоваться, чтобы остановить процесс восстановления в более ранней точке. Использоваться может только один из параметров `recovery_target`, `recovery_target_lsn`, `recovery_target_name`, `recovery_target_time` и `recovery_target_xid`; если в конфигурационном файле задано несколько параметров, будет использоваться последний.

`recovery_target= 'immediate'`

Данный параметр указывает, что процесс восстановления должен завершиться, как только будет достигнуто целостное состояние, т. е. как можно раньше. При восстановлении из оперативной резервной копии, это будет точкой, в которой завершился процесс резервного копирования.

Технически это строковый параметр, но значение `'immediate'` — единственно допустимое в данный момент.

`recovery_target_name` (`string`)

Этот параметр указывает именованную точку восстановления (созданную с помощью `pg_create_restore_point()`), до которой будет производиться восстановление.

`recovery_target_time` (`timestamp`)

Данный параметр указывает точку времени, вплоть до которой будет производиться восстановление. Точность этой точки останова также зависит от recovery_target_inclusive.

`recovery_target_xid` (`string`)

Параметр указывает идентификатор транзакции, вплоть до которой необходимо произвести процедуру восстановления. Имейте в виду, что несмотря на то, что при старте идентификаторы транзакций назначаются последовательно, завершаться они могут в ином порядке. Восстанавливаемые транзакции это те, что были зафиксированы до указанной (и, возможно, включая её). Точность точки останова также зависит от recovery_target_inclusive.

`recovery_target_lsn` (`pg_lsn`)

Данный параметр указывает LSN позиции в журнале предзаписи, до которой должно выполняться восстановление. Точная позиция остановки зависит также от параметра recovery_target_inclusive. Данный параметр принимает значение системного типа данных `pg_lsn`.

Следующие параметры уточняют целевую точку восстановления и определяют, что будет происходить при её достижении:

`recovery_target_inclusive` (`boolean`)

Указывает на необходимость остановки сразу после (`true`) либо до (`false`) достижения целевой точки. Применяется одновременно с recovery_target_lsn, recovery_target_time или recovery_target_xid. Этот параметр определяет, нужно ли восстанавливать транзакции, у которых позиция в WAL (LSN), время фиксации либо идентификатор в точности совпадает с заданным соответствующим значением. По умолчанию выбирается вариант `true`.

`recovery_target_timeline` (`string`)

Указывает линию времени для восстановления. По умолчанию производится восстановление той же линии времени, которая была текущей в момент создания базовой резервной копии. Со значением `latest` восстанавливаться будет последняя линия времени, найденная в архиве, что полезно для резервного сервера. Иное значение параметра может потребоваться в более сложной ситуации повторного восстановления, когда необходимо вернуться к состоянию, которое само было достигнуто после восстановления на момент времени. Это обсуждается в Подразделе 24.3.5.

`recovery_target_action` (`enum`)

Указывает, какое действие должен предпринять сервер после достижения цели восстановления. Вариант по умолчанию — `pause`, что означает приостановку восстановления. Второй вариант, `promote`, означает, что процесс восстановления завершится и сервер начнёт принимать подключения. Наконец, с вариантом `shutdown` сервер остановится, как только цель восстановления будет достигнута.

Вариант `pause` позволяет выполнить запросы к базе данных и убедиться в том, что достигнутая цель оказалась желаемой точкой восстановления. Для снятия с паузы нужно вызвать `pg_wal_replay_resume()` (см. Таблицу 9.81), что в итоге приведёт к завершению восстановления. Если же окажется, что мы ещё не достигли желаемой точки восстановления, нужно остановить сервер, установить более позднюю цель и перезапустить сервер для продолжения восстановления.

Вариант `shutdown` полезен для получения готового экземпляра сервера в желаемой точке. При этом данный экземпляр сможет воспроизводить дополнительные записи WAL (и на самом деле ему придётся воспроизводить записи WAL после последней контрольной точки при следующем перезапуске).

Заметьте, что так как `recovery.conf` не переименовывается, когда в `recovery_target_action` выбран вариант `shutdown`, при последующем запуске будет происходить немедленная остановка, пока вы не измените конфигурацию или не удалите файл `recovery.conf` вручную.

Этот параметр не действует, если цель восстановления не установлена. Если не включён режим hot_standby, значение `pause` действует так же, как и `shutdown`.

26.3. Параметры резервного сервера

`standby_mode` (`boolean`)

Указывает, является ли сервер Postgres Pro резервным или нет. Если параметр установлен в `on`, то сервер не прекратит восстановление по окончании последнего архивного файла WAL, а продолжит попытки извлечения новых сегментов WAL посредством команды `restore_command` и/или через подключение к ведущему, как указано в параметре `primary_conninfo`.

`primary_conninfo` (`string`)

Указывает строку подключения ведомого сервера к ведущему. Строка описана в формате согласно Подраздел 31.1.1. Вместо опущенных параметров подключения используются соответствующие переменные окружения (см. Раздел 31.14). Если же какие-то переменные не установлены, то используются значения по умолчанию.

Строка может содержать имя (или адрес) основного сервера и порт подключения. Также можно указать необходимого пользователя на ведущем сервере с требуемыми привилегиями (см.

Подраздел 25.2.5.1). Если настроена аутентификация по паролю, то его также необходимо указать. Пароль можно указать как в строке подключения `primary_conninfo`, так и в файле `~/.pgpass` ведомого сервера (в качестве имени базы данных используйте `replication`). Не указывайте имя базы данных среди параметров подключения `primary_conninfo`.

Этот параметр не действует, если `standby_mode` имеет значение `off`.

`primary_slot_name` (`string`)

Дополнительно задаёт заранее созданный слот, который будет использоваться при подключении к ведущему по протоколу потоковой репликации для управления освобождением ресурсов вышестоящего узла (см. Подраздел 25.2.6). Этот параметр не действует, если не указана строка подключения `primary_conninfo`.

`trigger_file` (`string`)

Указывает триггерный файл, при появлении которого завершается работа в режиме ведомого. Даже если это значение не установлено, существует возможность назначить ведомый сервер ведущим с помощью команды `pg_ctl promote`. Этот параметр не действует, если параметр `standby_mode` имеет значение `off`.

`recovery_min_apply_delay` (`integer`)

По умолчанию ведомый сервер восстанавливает записи WAL ведущего настолько быстро, насколько это возможно. Иногда полезно иметь возможность задать задержку при копировании данных, например, для устранения ошибок, связанных с потерей данных. Параметр позволяет задать эту задержку, указав период времени в миллисекундах (по умолчанию) или иных единицах измерения. Например, если установить значение `5min`, ведомый сервер будет воспроизводить фиксацию транзакции не раньше, чем через 5 минут (судя по его системным часам) после времени фиксации, сообщённого ведущим.

Возможна ситуация, когда задержка репликации между серверами превышает значение этого параметра. В этом случае дополнительная задержка не добавляется. Заметьте, что задержка вычисляется как разница между меткой времени, записанной в WAL на ведущем сервере, и текущим временем на ведомом. Запаздывание передачи, связанное с задержками в сети или каскадной репликацией, может существенно сократить реальное время ожидания. Если время на главном и ведомом сервере не синхронизировано, это может приводить к применению записей ранее ожидаемого, однако это не очень важно, потому что полезные значения этого параметра намного превышают типичное расхождение во времени между двумя серверами.

Задержка применяется лишь для записей WAL, представляющих фиксацию транзакций. Остальные записи проигрываются незамедлительно, так как их эффект не будет заметен до применения соответствующей записи о фиксации транзакции, благодаря правилам видимости MVCC.

Задержка добавляется, как только восстанавливаемая база данных достигает согласованного состояния, и исключается, когда ведущий сервер переключается в режим основного. После переключения ведущий сервер завершает восстановление незамедлительно.

Данный параметр предназначен для применения в конфигурациях с потоковой репликацией; однако если он задан, он будет учитываться в любых случаях. Задержка, устанавливаемая этим параметром, влияет и на работу механизма `hot_standby_feedback`, что может привести к раздуванию базы на главном сервере; использовать данный параметр при включении этого механизма следует с осторожностью.

> **Предупреждение**
>
> Этот параметр влияет на синхронную репликацию, когда `synchronous_commit` имеет значение `remote_apply`; каждый `COMMIT` будет ждать подтверждения применения.

Глава 27. Мониторинг работы СУБД

Администратор базы данных часто задумывается — «чем в данный момент занята система?» В этой главе рассказывается о том, как это выяснить.

Для мониторинга работы СУБД и анализа её производительности существуют различные инструменты. Большая часть этой главы посвящена описанию работы сборщика статистики Postgres Pro, однако не следует пренебрегать и обычными командами мониторинга Unix, такими как `ps`, `top`, `iostat`, и `vmstat`. Кроме того, после обнаружения запроса с низкой производительностью может потребоваться дополнительное исследование с использованием Postgres Pro команды EXPLAIN. В Разделе 14.1 рассматриваются `EXPLAIN` и другие методы для изучения поведения отдельного запроса.

27.1. Стандартные инструменты Unix

В большинстве Unix-платформ Postgres Pro модифицирует заголовок команды, который выводится на экран при выполнении команды `ps` так, что серверные процессы можно легко различить. Вот пример вывода этой команды:

```
$ ps auxww | grep ^postgres
postgres  15551  0.0  0.1  57536  7132 pts/0    S    18:02   0:00 postgres -i
postgres  15554  0.0  0.0  57536  1184 ?        Ss   18:02   0:00 postgres: writer
 process
postgres  15555  0.0  0.0  57536   916 ?        Ss   18:02   0:00 postgres:
 checkpointer process
postgres  15556  0.0  0.0  57536   916 ?        Ss   18:02   0:00 postgres: wal writer
 process
postgres  15557  0.0  0.0  58504  2244 ?        Ss   18:02   0:00 postgres: autovacuum
 launcher process
postgres  15558  0.0  0.0  17512  1068 ?        Ss   18:02   0:00 postgres: stats
 collector process
postgres  15582  0.0  0.0  58772  3080 ?        Ss   18:04   0:00 postgres: joe runbug
 127.0.0.1 idle
postgres  15606  0.0  0.0  58772  3052 ?        Ss   18:07   0:00 postgres: tgl
 regression [local] SELECT waiting
postgres  15610  0.0  0.0  58772  3056 ?        Ss   18:07   0:00 postgres: tgl
 regression [local] idle in transaction
```

(Формат вызова `ps`, а так же детали отображаемой информации зависят от платформы. Это пример для одной из последних Linux систем.) Первым здесь перечислен главный процесс сервера. Для этого процесса отображены аргументы команды, которые использовались при его запуске. Следующие пять процессов — это фоновые рабочие процессы, которые были автоматически запущены процессом сервера. (Процесса «stats collector» в этом списке не будет, если запуск сборщика статистики отключён в системе; аналогично может быть отключён и процесс «autovacuum launcher» — фоновый процесс автоочистки.) Во всех остальных строках перечислены серверные процессы, каждый из которых обрабатывает одно клиентское подключение. Командная строка каждого такого процесса имеет следующий формат:

`postgres: `*`user database host activity`*

Пользователь, СУБД и компьютер (клиента) остаются неизменными на протяжении всего клиентского подключения, а индикатор деятельности меняется. Возможные виды деятельности: `idle` (т. е. ожидание команды клиента), `idle in transaction` (ожидание клиента внутри блока `BEGIN`) или название типа команды, например, `SELECT`. Кроме того, если в настоящий момент серверный процесс ожидает высвобождения блокировки, которую держит другая сессия, то к виду деятельности добавляется `waiting`. В приведённом выше примере мы видим, что процесс 15606 ожидает, когда процесс 15610 завершит свою транзакцию и, следовательно, освободит какую-то блокировку. (Процесс 15610 является блокирующим, поскольку никаких других активных сессий нет. В более сложных случаях может потребоваться обращение к системному представлению `pg_locks`, для того чтобы определить, кто кого блокирует.)

Если установлено значение cluster_name, имя кластера также будет показываться в выводе команды ps:

```
$ psql -c 'SHOW cluster_name'
 cluster_name
--------------
 server1
(1 row)

$ ps aux|grep server1
postgres   27093  0.0  0.0  30096  2752 ?        Ss   11:34   0:00 postgres: server1:
 writer process
...
```

Если параметр update_process_title был отключён, то индикатор деятельности не обновляется; название процесса устанавливается только один раз при запуске нового процесса. На некоторых платформах это позволяет значительно сократить накладные расходы при выполнении команды; на других платформах этот выигрыш может быть незначителен.

> **Подсказка**
>
> В Solaris требуется особый подход. Следует использовать /usr/ucb/ps вместо /bin/ps. Также следует использовать два флага w, а не один. Кроме того, при выводе статусов команд с помощью ps статус для исходной команды postgres должен отображаться в сокращённом формате для каждого серверного процесса. Если вы не сделаете все три вещи, то вывод ps для каждого серверного процесса будет исходной командной строкой postgres.

27.2. Сборщик статистики

Сборщик статистики в Postgres Pro представляет собой подсистему, которая собирает и отображает информацию о работе сервера. В настоящее время сборщик может подсчитывать количество обращений к таблицам и индексам — в виде количества прочитанных блоков или строк с диска. Кроме того, он отслеживает общее число строк в каждой таблице, информацию о выполнении очистки и сбора статистики для каждой таблицы. Он также может подсчитывать вызовы пользовательских функций и общее время, затраченное на выполнение каждой из них.

Кроме того, Postgres Pro может предоставить динамическую информацию о том, что происходит в системе прямо сейчас, в частности, сообщить, какие именно команды выполняются другими серверными процессами и какие другие соединения существуют в системе. Эта возможность не зависит от процесса сборщика.

27.2.1. Конфигурация системы сбора статистики

Поскольку сбор статистики несколько увеличивает накладные расходы при выполнении запроса, есть возможность настроить СУБД так, чтобы выполнять или не выполнять сбор статистической информации. Это контролируется конфигурационными параметрами, которые обычно устанавливаются в файле postgresql.conf. (Подробно установка конфигурационных параметров описывается в Главе 18.)

Параметр track_activities включает мониторинг текущих команд, выполняемой любым серверным процессом.

Параметр track_counts определяет необходимость сбора статистики по обращениям к таблицам и индексам.

Параметр track_functions включает отслеживание использования пользовательских функций.

Параметр track_io_timing включает мониторинг времени чтения и записи блоков.

Обычно эти параметры устанавливаются в `postgresql.conf`, поэтому они применяются ко всем серверным процессам, однако, используя команду SET, их можно включать и выключать в отдельных сессиях. (Для того чтобы обычные пользователи не скрывали свою работу от администратора СУБД, изменять эти параметры с помощью команды `SET` могут только суперпользователи.)

Сборщик статистики использует временные файлы для передачи собранной информации другим процессам Postgres Pro. Имя каталога, в котором хранятся эти файлы, задаётся параметром stats_temp_directory, по умолчанию он называется `pg_stat_tmp`. Для повышения производительности `stats_temp_directory` может указывать на каталог, расположенный в оперативной памяти, что сокращает время физического ввода/вывода. При остановке сервера постоянная копия статистической информации сохраняется в подкаталоге `pg_stat`, поэтому статистику можно хранить на протяжении нескольких перезапусков сервера. Когда восстановление выполняется при запуске сервера (например, после непосредственного завершения работы, катастрофического отказа сервера, и восстановлении на заданную точку во времени), все статистические данные счётчиков сбрасываются.

27.2.2. Просмотр статистики

Для просмотра текущего состояния системы предназначены несколько предопределённых представлений, которые перечислены в Таблице 27.1. В дополнение к ним есть несколько других представлений, перечисленных в Таблице 27.2, позволяющих просмотреть результаты сбора статистики. Кроме того, на базе нижележащих статистических функций можно создать собственные представления, как описано в Подразделе 27.2.3.

Наблюдая собранные данные в сборщике статистики, важно понимать, что эта информация обновляется не сразу. Каждый серверный процесс передаёт новые статистические данные сборщику статистики непосредственно перед переходом в режим ожидания; то есть запрос или транзакция в процессе выполнения не влияют на отображаемые данные статистики. К тому же, сам сборщик статистики формирует новый отчёт не чаще, чем раз в `PGSTAT_STAT_INTERVAL` миллисекунд (500 мс, если этот параметр не изменялся при компиляции сервера). Так что отображаемая информация отстаёт от того, что происходит в настоящий момент. Однако информация о текущем запросе, собираемая с параметром `track_activities`, всегда актуальна.

Ещё одним важным моментом является то, что когда в серверном процессе запрашивают какую-либо статистику, сначала он получает наиболее свежий моментальный снимок от сборщика статистики и затем до окончания текущей транзакции использует этот снимок для всех статистических представлений и функций. Так что на протяжении одной транзакции статистическая информация меняться не будет. Подобным же образом информация о текущих запросах во всех сессиях собирается в тот момент, когда она впервые запрашивается в рамках транзакции, и эта же самая информация будет отображаться на протяжении всей транзакции. Это не ошибка, а полезное свойство СУБД, поскольку оно позволяет выполнять запросы к статистическим данным и сравнивать результаты, не беспокоясь о том, что статистические данные изменяются. Но если для каждого запроса вам нужны новые результаты, то их следует выполнять вне любых транзакционных блоков. Или же можно вызывать функцию `pg_stat_clear_snapshot`(), которая сбросит ранее полученный снимок статистики в текущей транзакции (если он был). При следующем обращении к статистической информации будет сформирован новый моментальный снимок.

Через представления `pg_stat_xact_all_tables`, `pg_stat_xact_sys_tables`, `pg_stat_xact_user_tables`, и `pg_stat_xact_user_functions` транзакции также доступна её собственная статистика (ещё не переданная сборщику статистики). Данные в этих представлениях ведут себя не так, как описано выше; наоборот, в течение транзакции они постоянно обновляются.

Таблица 27.1. Динамические статистические представления

Имя представления	Описание
`pg_stat_activity`	Одна строка для каждого серверного процесса с информацией по текущей активности процесса,

Имя представления	Описание
	такой как состояние и текущий запрос. За подробностями обратитесь к pg_stat_activity.
pg_stat_replication	По одной строке для каждого процесса-передатчика WAL со статистикой по репликации на ведомом сервере, к которому подключён этот процесс. За подробностями обратитесь к pg_stat_replication.
pg_stat_wal_receiver	Только одна строка со статистикой приёмника WAL, полученной с сервера, на котором работает приёмник. За подробностями обратитесь к pg_stat_wal_receiver.
pg_stat_subscription	Как минимум одна строка для подписки, сообщающая о рабочих процессах подписки. За подробностями обратитесь к pg_stat_subscription.
pg_stat_ssl	Одна строка для каждого подключения (обычного и реплицирующего), в которой показывается информация об использовании SSL для данного подключения. За подробностями обратитесь к pg_stat_ssl.
pg_stat_progress_vacuum	По одной строке с текущим состоянием для каждого обслуживающего процесса (включая рабочие процессы автоочистки), в котором работает VACUUM. См. Подраздел 27.4.1.

Таблица 27.2. Представления собранной статистики

Имя представления	Описание
pg_stat_archiver	Только одна строка со статистикой о работе активности процесса архивации WAL. Более подробно смотрите pg_stat_archiver.
pg_stat_bgwriter	Только одна строка со статистикой о работе фонового процесса записи. Более подробно смотрите pg_stat_bgwriter.
pg_stat_database	Одна строка для каждой базы данных со статистикой на уровне базы. Более подробно смотрите pg_stat_database.
pg_stat_database_conflicts	По одной строке на каждую базу данных со статистикой по отменам запросов, выполненным вследствие конфликта с процессами восстановления на ведомых серверах. Более подробно смотрите pg_stat_database_conflicts.
pg_stat_all_tables	По одной строке на каждую таблицу в текущей базе данных со статистикой по обращениям к этой таблице. Более подробно смотрите pg_stat_all_tables.
pg_stat_sys_tables	Аналогично pg_stat_all_tables , за исключением того, что отображаются только системные таблицы.

Имя представления	Описание
pg_stat_user_tables	Аналогично pg_stat_all_tables, за исключением того, что отображаются только пользовательские таблицы.
pg_stat_xact_all_tables	Подобно pg_stat_all_tables, но подсчитывает действия, выполненные в текущей транзакции к настоящему моменту (которые ещё *не* вошли в pg_stat_all_tables и связанные представления). Столбцы для числа живых и мёртвых строк, а также количества операций очистки и сбора статистики, в этом представлении отсутствуют.
pg_stat_xact_sys_tables	Аналогично pg_stat_xact_all_tables, за исключением того, что отображаются только системные таблицы.
pg_stat_xact_user_tables	Аналогично pg_stat_xact_all_tables, за исключением того, что отображаются только пользовательские таблицы.
pg_stat_all_indexes	По одной строке для каждого индекса в текущей базе данных со статистикой по обращениям к этому индексу. Более подробно смотрите pg_stat_all_indexes.
pg_stat_sys_indexes	Аналогично pg_stat_all_indexes, за исключением того, что показываются только индексы по системным таблицам.
pg_stat_user_indexes	Аналогично pg_stat_all_indexes, за исключением того, что показываются только индексы по пользовательским таблицам.
pg_statio_all_tables	По одной строке для каждой таблицы в текущей базе данных со статистикой по операциям ввода/вывода для этой таблицы. Более подробно смотрите pg_statio_all_tables.
pg_statio_sys_tables	Аналогично pg_statio_all_tables, за исключением того, что показываются только системные таблицы.
pg_statio_user_tables	Аналогично pg_statio_all_tables, за исключением того, что показываются только пользовательские таблицы.
pg_statio_all_indexes	По одной строке для каждого индекса в текущей базе данных со статистикой по операциям ввода/вывода для этого индекса. Более подробно смотрите pg_statio_all_indexes.
pg_statio_sys_indexes	Аналогично pg_statio_all_indexes, за исключением того, что показываются только индексы по системным таблицам.
pg_statio_user_indexes	Аналогично pg_statio_all_indexes, за исключением того, что показываются только индексы по пользовательским таблицам.
pg_statio_all_sequences	По одной строке для каждой последовательности в текущей базе данных со статистикой по операциям ввода/вывода

Имя представления	Описание
	для этой последовательности. Более подробно смотрите pg_statio_all_sequences.
pg_statio_sys_sequences	Аналогично pg_statio_all_sequences , за исключением того, что показываются только системные последовательности. (В настоящее время системных последовательностей нет, поэтому это представление всегда пусто.)
pg_statio_user_sequences	Аналогично pg_statio_all_sequences , за исключением того, что показываются только пользовательские последовательности.
pg_stat_user_functions	По одной строке для каждой отслеживаемой функции со статистикой по выполнениям этой функции. Более подробно смотрите pg_stat_user_functions.
pg_stat_xact_user_functions	Аналогично pg_stat_user_functions , однако подсчитываются только вызовы функций, выполненные в текущей транзакции (которые ещё *не* были включены в pg_stat_user_functions).

Статистика по отдельным индексам особенно полезна для определения того, какие индексы используются и насколько они эффективны.

Представления pg_statio_ полезны, прежде всего, для определения эффективности буферного кеша. Если количество фактических дисковых чтений существенно меньше количества чтений из буферного кеша, то это означает, что кеш справляется с большинством запросов на чтение без обращения к ядру. Однако эта статистика не даёт полной картины: Postgres Pro обрабатывает дисковый ввод/вывод так, что данные, не находящиеся в буферном кеше Postgres Pro, могут все ещё располагаться в кеше ввода/вывода ядра, и, следовательно, для их получения физическое чтение может не использоваться. Для получения более детальной информации о процессе ввода/вывода в Postgres Pro рекомендуется использовать сборщик статистики Postgres Pro в сочетании с утилитами операционной системы, которые дают более полное представление о том, как ядро осуществляет ввод/вывод.

Таблица 27.3. Представление pg_stat_activity

Столбец	Тип	Описание
datid	oid	OID базы данных, к которой подключён этот серверный процесс
datname	name	Имя базы данных, к которой подключён этот серверный процесс
pid	integer	Идентификатор процесса этого серверного процесса
usesysid	oid	OID пользователя, подключённого к этому серверному процессу
usename	name	Имя пользователя, подключённого к этому серверному процессу
application_name	text	Название приложения, подключённого к этому серверному процессу

Столбец	Тип	Описание
client_addr	inet	IP-адрес клиента, подключённого к этому серверному процессу. Значение null в этом поле означает, что клиент подключён через сокет Unix на стороне сервера или что это внутренний процесс, например, автоочистка.
client_hostname	text	Имя компьютера для подключённого клиента, получаемое в результате обратного поиска в DNS по client_addr . Это поле будет отлично от null только в случае соединений по IP и только при включённом режиме log_hostname.
client_port	integer	Номер TCP-порта, который используется клиентом для соединения с этим серверным процессом, или -1, если используется сокет Unix
backend_start	timestamp with time zone	Время запуска процесса. Для процессов, обслуживающих клиентов, это время подключения клиента к серверу.
xact_start	timestamp with time zone	Время начала текущей транзакции в этом процессе или null при отсутствии активной транзакции. Если текущий запрос был первым в своей транзакции, то значение в этом столбце совпадает со значением столбца query_start .
query_start	timestamp with time zone	Время начала выполнения активного в данный момент запроса, или, если state не active, то время начала выполнения последнего запроса
state_change	timestamp with time zone	Время последнего изменения состояния (поля state)
wait_event_type	text	Тип события, которого ждёт обслуживающий процесс, если это имеет место; в противном случае — NULL. Возможные значения: • LWLock: обслуживающий процесс ожидает лёгкую блокировку. Такие блокировки защищают

Столбец	Тип	Описание
		определённые структуры данных в разделяемой памяти. В `wait_event` будет содержаться имя, отражающее цель получения лёгкой блокировки. (Некоторые блокировки имеют особые имена; другие объединяются в группы блокировок со схожим предназначением.) • `Lock`: Обслуживающий процесс ожидает тяжёлую блокировку. Тяжёлые блокировки, также называемые блокировками менеджера блокировок или просто блокировками, в основном защищают объекты уровня SQL, такие как таблицы. Однако они также применяются для взаимоисключающего выполнения некоторых внутренних операций, например, для расширения отношений. Тип ожидаемой блокировки показывается в `wait_event` . • `BufferPin`: Серверный процесс ожидает доступа к буферу данных, когда никакой другой процесс не обращается к этому буферу. Ожидание закрепления буфера может растягиваться, если другой процесс удерживает открытый курсор, который читал данные из нужного буфера. • `Activity`: Серверный процесс простаивает. Это состояние наблюдается в системных процессах, ожидающих активности в основном цикле обработки. В `wait_event` обозначается конкретное место ожидания. • `Extension`: Серверный процесс ожидает активности в модуле расширения. Эта категория полезна при

Столбец	Тип	Описание
		использовании модулей, она помогает отслеживать нестандартные места ожидания. • `Client`: Серверный процесс ожидает в сокете некоторую активность пользовательского приложения, и события, ожидаемые сервером, не зависят от его внутренних процессов. В `wait_event` обозначается конкретное место ожидания. • `IPC`: Серверный процесс ожидает некоторой активности другого процесса на сервере. В `wait_event` обозначается конкретное место ожидания. • `Timeout`: Серверный процесс ожидает истечения определённого времени. В `wait_event` обозначается конкретное место ожидания. • `IO`: Серверный процесс ожидает завершения операции ввода/вывода. В `wait_event` обозначается конкретное место ожидания.
`wait_event`	`text`	Имя ожидаемого события, если обслуживающий процесс находится в состоянии ожидания, а в противном случае — NULL. За подробностями обратитесь к Таблице 27.4.
`state`	`text`	Общее текущее состояние этого серверного процесса. Возможные значения: • `active`: серверный процесс выполняет запрос. • `idle`: серверный процесс ожидает новой команды от клиента. • `idle in transaction`: серверный процесс находится внутри транзакции, но в настоящее

Столбец	Тип	Описание
		время не выполняет никакой запрос. • `idle in transaction (aborted)` : Это состояние подобно `idle in transaction`, за исключением того, что один из операторов в транзакции вызывал ошибку. • `fastpath function call`: серверный процесс выполняет fast-path функцию. • `disabled`: Это состояние отображается для серверных процессов, у которых параметр track_activities отключён.
`backend_xid`	`xid`	Идентификатор верхнего уровня транзакции этого серверного процесса или любой другой.
`backend_xmin`	`xid`	текущая граница `xmin` для серверного процесса.
`query`	`text`	Текст последнего запроса этого серверного процесса. Если `state` имеет значение `active`, то в этом поле отображается запрос, который выполняется в настоящий момент. Если процесс находится в любом другом состоянии, то в этом поле отображается последний выполненный запрос. По умолчанию текст запроса обрезается до 1024 байт; это число определяется параметром track_activity_query_size.
`backend_type`	`text`	Тип текущего серверного процесса. Возможные варианты: `autovacuum launcher`, `autovacuum worker`, `background worker`, `background writer`, `client backend`, `checkpointer`, `startup`, `walreceiver`, `walsender` и `walwriter`.

В представлении `pg_stat_activity` для каждого серверного процесса будет присутствовать по одной строке с информацией, относящейся к текущей деятельности этого процесса.

Примечание

Значения в столбцах `wait_event` и `state` не зависят друг от друга. Если обслуживающий процесс находится в состоянии `active` (активен), он может ожидать какое-то событие, или не ожидать никакое. Если состояние `active` и поле `wait_event` содержит не NULL, это означает, что запрос выполняется, но заблокирован чем-то в системе.

Таблица 27.4. Описание `wait_event`

Тип события ожидания	Название события ожидания	Описание
LWLock	ShmemIndexLock	Ожидание при поиске или выделении области в разделяемой памяти.
	OidGenLock	Ожидание при выделении или назначении OID.
	XidGenLock	Ожидание при выделении или назначении идентификатора транзакции.
	ProcArrayLock	Ожидание при получении снимка или очистке идентификатора транзакции в конце транзакции.
	SInvalReadLock	Ожидание при получении или удалении из общей очереди сообщений аннулирования.
	SInvalWriteLock	Ожидание при добавлении в общую очередь сообщения аннулирования.
	WALBufMappingLock	Ожидание при замене страницы в буферах WAL.
	WALWriteLock	Ожидание при записи буферов WAL на диск.
	ControlFileLock	Ожидание при чтении или изменении управляющего файла либо при создании нового файла WAL.
	CheckpointLock	Ожидание при выполнении контрольной точки.
	CLogControlLock	Ожидание при чтении или изменении состояния транзакции.
	SubtransControlLock	Ожидание при чтении или изменении информации о подтранзакции.
	MultiXactGenLock	Ожидание при чтении или изменении общего состояния мультитранзакций.
	MultiXactOffsetControlLock	Ожидание при чтении или изменении смещений мультитранзакций.

Тип события ожидания	Название события ожидания	Описание
	MultiXactMemberControlLock	Ожидание при чтении или изменении членов мультитранзакций.
	RelCacheInitLock	Ожидание при чтении или записи файла инициализации кеша отношения.
	CheckpointerCommLock	Ожидание при управлении запросами fsync.
	TwoPhaseStateLock	Ожидание при чтении или изменении состояния подготовленных транзакций.
	TablespaceCreateLock	Ожидание при создании или удалении табличного пространства.
	BtreeVacuumLock	Ожидание при чтении или изменении информации, связанной с очисткой, для индекса-B-дерева.
	AddinShmemInitLock	Ожидание при управлении выделением блоков в общей памяти.
	AutovacuumLock	Ожидание в рабочем процессе или процедуре запуска автоочистки при изменении или чтении текущего состояния рабочих процессов автоочистки.
	AutovacuumScheduleLock	Ожидание при подтверждении, что таблица, выбранная для очистки, всё ещё нуждается в очистке.
	SyncScanLock	Ожидание при получении начального положения сканирования таблицы для синхронизированного сканирования.
	RelationMappingLock	Ожидание при изменении в файле сопоставления отношений, используемого для хранения связей файловых узлов с каталогом БД.
	AsyncCtlLock	Ожидание при чтении или изменении общего состояния уведомлений.
	AsyncQueueLock	Ожидание при чтении или изменении сообщений уведомлений.
	SerializableXactHashLock	Ожидание при получении или сохранении информации о сериализуемых транзакциях.

Тип события ожидания	Название события ожидания	Описание
	SerializableFinishedListLock	Ожидание при обращении к списку завершённых сериализуемых транзакций.
	SerializablePredicateLockListLock	Ожидание при выполнении операции со списком блокировок, удерживаемых сериализуемыми транзакциями.
	OldSerXidLock	Ожидание при чтении или записи информации о конфликтующих сериализуемых транзакциях.
	SyncRepLock	Ожидание при чтении или изменении сведений о синхронных репликах.
	BackgroundWorkerLock	Ожидание при чтении или изменении состояния фонового рабочего процесса.
	DynamicSharedMemoryControlLock	Ожидание при чтении или изменении состояния динамической общей памяти.
	AutoFileLock	Ожидание при изменении файла postgresql.auto.conf.
	ReplicationSlotAllocationLock	Ожидание при выделении или освобождении слота репликации.
	ReplicationSlotControlLock	Ожидание при чтении или изменении состояния слота репликации.
	CommitTsControlLock	Ожидание при чтении или изменении времени фиксирования транзакции.
	CommitTsLock	Ожидание при чтении или изменении последнего значения, заданного в качестве времени транзакции.
	ReplicationOriginLock	Ожидание при подготовке, удалении или использовании источника репликации.
	MultiXactTruncationLock	Ожидание при чтении или очистке информации мультитранзакций.
	OldSnapshotTimeMapLock	Ожидание при чтении или изменении информации о старом снимке.
	BackendRandomLock	Ожидание генерирования случайного числа.

Тип события ожидания	Название события ожидания	Описание
	LogicalRepWorkerLock	Ожидание завершения действия процесса логической репликации.
	CLogTruncationLock	Ожидание выполнения функции txid_status или обновления самого старого видимого в ней идентификатора транзакции.
	WrapLimitsVacuumLock	Ожидание при изменении лимитов идентификаторов транзакций и мультитранзакций.
	NotifyQueueTailLock	Ожидание при изменении границы массива с сообщениями уведомлений.
	clog	Ожидание при вводе/выводе с буфером clog (буфер состояния транзакций).
	commit_timestamp	Ожидание при вводе/выводе с буфером времени фиксирования транзакций.
	subtrans	Ожидание при вводе/выводе с буфером подтранзакций.
	multixact_offset	Ожидание при вводе/выводе с буфером смещений мультитранзакций.
	multixact_member	Ожидание при вводе/выводе с буфером multixact_member.
	async	Ожидание при вводе/выводе с буфером асинхронных сообщений (уведомлений).
	oldserxid	Ожидание при вводе/выводе с буфером oldserxid.
	wal_insert	Ожидание при добавлении записей WAL в буфер в памяти.
	buffer_content	Ожидание при чтении или записи страницы данных в памяти.
	buffer_io	Ожидание при вводе/выводе, связанном со страницей данных.
	replication_origin	Ожидание при чтении или изменении состояния репликации.
	replication_slot_io	Ожидание при вводе/выводе со слотом репликации.
	proc	Ожидание при чтении или изменении информации о блокировках по быстрому пути.

Тип события ожидания	Название события ожидания	Описание
	buffer_mapping	Ожидание при связывании блока данных с буфером в пуле буферов.
	lock_manager	Ожидание при добавлении или обращении к блокировкам обслуживающих процессов либо ожидание входа или выхода из группы блокировок (используется в параллельных запросах).
	predicate_lock_manager	Ожидание при добавлении или обращении к информации о предикатных блокировках.
	parallel_query_dsa	Ожидание блокировки выделения динамической общей памяти для параллельного запроса.
	tbm	Ожидание блокировки общего итератора TBM.
Lock	relation	Ожидание при запросе блокировки для отношения.
	extend	Ожидание при расширении отношения.
	frozenid	Ожидание изменения pg_database.datfrozenxid и pg_database.datminmxid.
	page	Ожидание при запросе блокировки для страницы отношения.
	tuple	Ожидание при запросе блокировки для кортежа.
	transactionid	Ожидание завершения транзакции.
	virtualxid	Ожидание при запросе блокировки виртуального xid.
	speculative token	Ожидание при запросе блокировки спекулятивного добавления.
	object	Ожидание при запросе блокировки для нереляционного объекта БД.
	userlock	Ожидание при запросе пользовательской блокировки.
	advisory	Ожидание при запросе рекомендательной пользовательской блокировки.
BufferPin	BufferPin	Ожидание при закреплении буфера.

Тип события ожидания	Название события ожидания	Описание
Activity	ArchiverMain	Ожидание в основном цикле процесса архиватора.
	AutoVacuumMain	Ожидание в основном цикле процесса запуска автоочистки.
	BgWriterHibernate	Ожидание в фоновом процессе записи, переход в режим «заморозки».
	BgWriterMain	Ожидание в основном цикле рабочего процесса фоновой записи.
	CheckpointerMain	Ожидание в основном цикле процесса контрольной точки.
	LogicalApplyMain	Ожидание в основном цикле процесса применения логической репликации.
	LogicalLauncherMain	Ожидание в основном цикле процесса запуска обработчиков логической репликации.
	PgStatMain	Ожидание в основном цикле процесса сборщика статистики.
	RecoveryWalAll	Ожидание поступления WAL из потока при восстановлении.
	RecoveryWalStream	Ожидание перед повторной попыткой получения WAL из любого источника (локального, архива или потока) при восстановлении.
	SysLoggerMain	Ожидание в основном цикле процесса системного журнала (syslogger).
	WalReceiverMain	Ожидание в основном цикле процесса-приёмника WAL.
	WalSenderMain	Ожидание в основном цикле процесса-передатчика WAL.
	WalWriterMain	Ожидание в основном цикле процесса, пишущего WAL.
Client	ClientRead	Ожидание при чтении данных, получаемых от клиента.
	ClientWrite	Ожидание при записи данных, передаваемых клиенту.
	LibPQWalReceiverConnect	Ожидание в приёмнике WAL установления подключения к удалённому серверу.
	LibPQWalReceiverReceive	Ожидание в приёмнике WAL поступления данных от удалённого сервера.
	SSLOpenServer	Ожидание SSL при попытке установления соединения.

Тип события ожидания	Название события ожидания	Описание
	WalReceiverWaitStart	Ожидание от стартового процесса передачи начальных данных для потоковой репликации.
	WalSenderWaitForWAL	Ожидание сброса WAL в процессе-передатчике WAL.
	WalSenderWriteData	Ожидание какой-либо активности при обработке ответов от WAL-приёмника в процессе-передатчике WAL.
Extension	Extension	Ожидание в расширении.
IPC	BgWorkerShutdown	Ожидание завершения фонового рабочего процесса.
	BgWorkerStartup	Ожидание запуска фонового рабочего процесса.
	BtreePage	Ожидание доступности номера страницы, необходимого для продолжения параллельного сканирования B-дерева.
	ExecuteGather	Ожидание активности дочернего процесса при выполнении узла Gather.
	LogicalSyncData	Ожидание от удалённого сервера логической репликации передачи данных для начальной синхронизации таблиц.
	LogicalSyncStateChange	Ожидание изменения состояния удалённого сервера логической репликации.
	MessageQueueInternal	Ожидание подключения другого процесса к общей очереди сообщений.
	MessageQueuePutMessage	Ожидание записи сообщения протокола в общую очередь сообщений.
	MessageQueueReceive	Ожидание получения байтов из общей очереди сообщений.
	MessageQueueSend	Ожидание передачи байтов в общую очередь сообщений.
	ParallelBitmapScan	Ожидание инициализации параллельного сканирования по битовой карте.
	ParallelFinish	Ожидание завершения вычислений параллельными рабочими процессами.
	ProcArrayGroupUpdate	Ожидание очистки ведущим группы идентификатора

Тип события ожидания	Название события ожидания	Описание
		транзакции в конце транзакции.
	ReplicationOriginDrop	Ожидание перехода источника репликации в неактивное состояние для последующего удаления.
	ReplicationSlotDrop	Ожидание перехода слота репликации в неактивное состояние для последующего удаления.
	SafeSnapshot	Ожидание снимка для транзакции READ ONLY DEFERRABLE.
	SyncRep	Ожидание подтверждения от удалённого сервера при синхронной репликации.
Timeout	BaseBackupThrottle	Ожидание в процессе базового резервного копирования из-за ограничения активности.
	PgSleep	Ожидание в процессе, вызвавшем pg_sleep .
	RecoveryApplyDelay	Ожидание применения WAL при восстановлении из-за задержки.
IO	BufFileRead	Ожидание чтения из буферизованного файла.
	BufFileWrite	Ожидание записи в буферизованный файл.
	ControlFileRead	Ожидание чтения из управляющего файла.
	ControlFileSync	Ожидание помещения управляющего файла в надёжное хранилище.
	ControlFileSyncUpdate	Ожидание переноса изменений управляющего файла в надёжное хранилище.
	ControlFileWrite	Ожидание записи в управляющий файл.
	ControlFileWriteUpdate	Ожидание записи для изменения управляющего файла.
	CopyFileRead	Ожидание чтения во время операции копирования файла.
	CopyFileWrite	Ожидание записи во время операции копирования файла.
	DataFileExtend	Ожидание расширения файла данных отношения.

Тип события ожидания	Название события ожидания	Описание
	DataFileFlush	Ожидание помещения файла данных отношения в надёжное хранилище.
	DataFileImmediateSync	Ожидание немедленной синхронизации файла данных отношения с надёжным хранилищем.
	DataFilePrefetch	Ожидание асинхронной предвыборки из файла данных отношения.
	DataFileRead	Ожидание чтения из файла данных отношения.
	DataFileSync	Ожидание переноса изменений в файле данных отношения в надёжное хранилище.
	DataFileTruncate	Ожидание усечения файла данных отношения.
	DataFileWrite	Ожидание записи в файл данных отношения.
	DSMFillZeroWrite	Ожидание записи нулевых байт в файл, поддерживающий работу динамической общей памяти.
	LockFileAddToDataDirRead	Ожидание чтения при добавлении строки в файл блокировки каталога данных.
	LockFileAddToDataDirSync	Ожидание помещения данных в надёжное хранилище при добавлении строки в файл блокировки каталога данных.
	LockFileAddToDataDirWrite	Ожидание записи при добавлении строки в файл блокировки каталога данных.
	LockFileCreateRead	Ожидание чтения при создании файла блокировки каталога данных.
	LockFileCreateSync	Ожидание помещения данных в надёжное хранилище при создании файла блокировки каталога данных.
	LockFileCreateWrite	Ожидание записи при создании файла блокировки каталога данных.
	LockFileReCheckDataDirRead	Ожидание чтения во время перепроверки файла блокировки каталога данных.
	LogicalRewriteCheckpointSync	Ожидание помещения отображений логической перезаписи в надёжное

Тип события ожидания	Название события ожидания	Описание
		хранилище во время контрольной точки.
	LogicalRewriteMappingSync	Ожидание помещения данных отображений в надёжное хранилище в процессе логической перезаписи.
	LogicalRewriteMappingWrite	Ожидание записи данных отображений в процессе логической перезаписи.
	LogicalRewriteSync	Ожидание помещения отображений логической перезаписи в надёжное хранилище.
	LogicalRewriteTruncate	Ожидание усечения файла отображений в процессе логической перезаписи.
	LogicalRewriteWrite	Ожидание сохранения отображений логической перезаписи.
	RelationMapRead	Ожидание чтения файла отображений отношений.
	RelationMapSync	Ожидание помещения файла отображений отношений в надёжное хранилище.
	RelationMapWrite	Ожидание записи в файл отображений отношений.
	ReorderBufferRead	Ожидание чтения при работе с буфером переупорядочивания.
	ReorderBufferWrite	Ожидание записи при работе с буфером переупорядочивания.
	ReorderLogicalMappingRead	Ожидание чтения логического отображения при работе с буфером переупорядочивания.
	ReplicationSlotRead	Ожидание чтения из управляющего файла слота репликации.
	ReplicationSlotRestoreSync	Ожидание помещения в надёжное хранилище управляющего файла слота репликации при восстановлении его в памяти.
	ReplicationSlotSync	Ожидание помещения в надёжное хранилище управляющего файла слота репликации.
	ReplicationSlotWrite	Ожидание записи в управляющий файл слота репликации.

Тип события ожидания	Название события ожидания	Описание
	SLRUFlushSync	Ожидание помещения данных SLRU в надёжное хранилище во время контрольной точки или отключения базы данных.
	SLRURead	Ожидание чтения страницы SLRU.
	SLRUSync	Ожидание помещения данных SLRU в надёжное хранилище после записи страницы.
	SLRUWrite	Ожидание записи страницы SLRU.
	SnapbuildRead	Ожидание чтения сериализованного исторического снимка каталога БД.
	SnapbuildSync	Ожидание помещения сериализованного исторического снимка каталога БД в надёжное хранилище.
	SnapbuildWrite	Ожидание записи сериализованного исторического снимка каталога БД.
	TimelineHistoryFileSync	Ожидание помещения в надёжное хранилище файла истории линии времени, полученного через потоковую репликацию.
	TimelineHistoryFileWrite	Ожидание записи файла истории линии времени, полученного через потоковую репликацию.
	TimelineHistoryRead	Ожидание чтения файла истории линии времени.
	TimelineHistorySync	Ожидание помещения в надёжное хранилище только что созданного файла истории линии времени.
	TimelineHistoryWrite	Ожидание записи только что созданного файла истории линии времени.
	TwophaseFileRead	Ожидание чтения файла двухфазного состояния.
	TwophaseFileSync	Ожидание помещения файла двухфазного состояния в надёжное хранилище.
	TwophaseFileWrite	Ожидание записи файла двухфазного состояния.

Тип события ожидания	Название события ожидания	Описание
	WALBootstrapSync	Ожидание помещения WAL в надёжное хранилище в процессе начальной загрузки.
	WALBootstrapWrite	Ожидание записи страницы WAL в процессе начальной загрузки.
	WALCopyRead	Ожидание чтения при создании нового сегмента WAL путём копирования существующего.
	WALCopySync	Ожидание помещения в надёжное хранилище нового сегмента WAL, созданного путём копирования существующего.
	WALCopyWrite	Ожидание записи при создании нового сегмента WAL путём копирования существующего.
	WALInitSync	Ожидание помещения в надёжное хранилище нового инициализированного файла WAL.
	WALInitWrite	Ожидание записи при инициализации нового файла WAL.
	WALRead	Ожидание чтения из файла WAL.
	WALSenderTimelineHistoryRead	Ожидание чтения из файла истории линии времени при обработки процессом walsender команды timeline.
	WALSyncMethodAssign	Ожидание помещения данных в надёжное хранилище при назначении метода синхронизации WAL.
	WALWrite	Ожидание записи в файл WAL.

Примечание

Для траншей, регистрируемых расширениями, в поле `wait_event` выводится имя, указываемое расширением. Пользователь вполне может зарегистрировать транш и в обслуживающем процессе (воспользовавшись динамической общей памятью), в результате чего другие процессы не получат доступа к этой информации; в таких случаях в этом поле выводится `extension`.

Следующая команда показывает, как можно просмотреть события ожидания:

```
SELECT pid, wait_event_type, wait_event FROM pg_stat_activity WHERE wait_event is NOT
 NULL;
 pid  | wait_event_type |  wait_event
------+-----------------+--------------
 2540 | Lock            | relation
```

```
 6644 | LWLock          | ProcArrayLock
(2 rows)
```

Таблица 27.5. Представление `pg_stat_replication`

Столбец	Тип	Описание
`pid`	`integer`	Идентификатор процесса-передатчика WAL
`usesysid`	`oid`	OID пользователя, подключённого к этому процессу-передатчику WAL
`usename`	`name`	Имя пользователя, подключённого к этому процессу-передатчику WAL
`application_name`	`text`	Имя приложения, которое подключено к этому процессу-передатчику WAL
`client_addr`	`inet`	IP-адрес клиента, подключённого к этому процессу-передатчику WAL. Значение null в этом поле говорит о том, что клиент подсоединён через сокет Unix на сервере.
`client_hostname`	`text`	Имя компьютера для подключённого клиента, получаемое в результате обратного поиска в DNS по `client_addr` . Это поле будет отлично от null только в случае соединений по IP и только при включённом режиме log_hostname.
`client_port`	`integer`	Номер TCP-порта, который используется клиентом для взаимодействия с процессом-передатчиком WAL, или `-1`, если используется сокет Unix
`backend_start`	`timestamp with time zone`	Время запуска процесса, т. е. время подключения клиента к этому процессу-передатчику WAL
`backend_xmin`	`xid`	Значение `xmin`, полученное от ведомого сервера при включённом hot_standby_feedback.
`state`	`text`	Текущее состояние процесса-передатчика WAL. Возможные значения: • `startup`: Передатчик WAL запускается. • `catchup`: Подключённый к этому передатчику WAL

Столбец	Тип	Описание
		ведомый сервер догоняет ведущий. • `streaming`: Передатчик WAL транслирует изменения после того, как подключённый к нему ведомый сервер нагнал ведущий. • `backup`: Передатчик WAL передаёт резервную копию. • `stopping`: Передатчик WAL останавливается.
`sent_lsn`	`pg_lsn`	Последняя позиция в журнале предзаписи, переданная через это соединение
`write_lsn`	`pg_lsn`	Последняя позиция в журнале предзаписи, записанная на диск этим ведомым сервером
`flush_lsn`	`pg_lsn`	Последняя позиция в журнале предзаписи, сброшенная на диск этим ведомым сервером
`replay_lsn`	`pg_lsn`	Последняя позиция в журнале предзаписи, воспроизведённая в базе данных этим ведомым сервером
`write_lag`	`interval`	Время, прошедшее с момента локального сброса последних данных WAL до получения уведомления о том, что этот ведомый сервер записал их (но ещё не сбросил на диск и не применил). Это позволяет оценить задержку, возникающую при фиксации транзакции, когда в `synchronous_commit` выбран уровень `remote_write` , если данный сервер будет настроен как синхронный ведомый.
`flush_lag`	`interval`	Время, прошедшее с момента локального сброса последних данных WAL до получения уведомления о том, что этот ведомый сервер записал и сбросил их на диск (но ещё не применил). Это позволяет оценить задержку, возникающую при фиксации транзакции, когда в `synchronous_commit` выбран уровень `on`, если данный сервер

Столбец	Тип	Описание
		будет настроен как синхронный ведомый.
replay_lag	interval	Время, прошедшее с момента локального сброса последних данных WAL до получения уведомления о том, что этот ведомый сервер записал, сбросил на диск и применил их. Это позволяет оценить задержку, возникающую при фиксации транзакции, когда в synchronous_commit выбран уровень remote_apply , если данный сервер будет настроен как синхронный ведомый.
sync_priority	integer	Приоритет этого ведомого сервера для выбора в качестве синхронного ведомого при синхронной репликации с учётом приоритетов. При синхронной репликации с учётом кворума не имеет значения.
sync_state	text	Состояние синхронизации этого ведомого сервера. Возможные значения: • async: Этот ведомый сервер является асинхронным. • potential: Этот ведомый сервер сейчас является асинхронным, но может стать синхронным в случае отказа одного из текущих синхронных серверов. • sync: Этот ведомый сервер является синхронным. • quorum: Этот ведомый сервер считается кандитатом на участие в кворуме.

Представление pg_stat_replication для каждого процесса-передатчика WAL будет содержать по одной строке со статистикой о репликации на ведомый сервер, к которому подключён этот процесс. В представлении перечисляются только ведомые серверы, подключённые напрямую; информация о ведомых серверах, подключённых опосредованно, не представлена.

Длительность задержек, показываемая в представлении pg_stat_replication, включает время, которое потребовалось для того, чтобы записать, сбросить на диск и воспроизвести последние записи WAL и для того, чтобы передатчик WAL узнал об этом. Эта длительность отражает задержку фиксации, которая была (или могла быть) добавлена на уровнях синхронной фиксации, если ведомый сервер был настроен как синхронный. Для асинхронного ведомого в столбце replay_lag показывается примерная задержка перед тем, как последние транзакции становятся видны для запросов. Если ведомый сервер нагоняет передающий и в WAL отсутствует активность, последние длительности задержек будут отображаться ещё некоторое время, а затем сменятся на NULL.

Длительность задержек автоматически определяется при физической репликации. Модули логического декодирования могут не выдавать необходимые контрольные сообщения; в их отсутствие механизм отслеживания просто выводит задержку NULL.

> **Примечание**
>
> Выдаваемые длительности задержек не являются предсказанием времени, которое потребуется ведомому серверу, чтобы нагнать передающий сервер с учётом текущей скорости воспроизведения. Эти показатели будут близки в процессе генерирования нового WAL, но не в то время, когда передающий сервер будет простаивать. В частности, когда ведомый сервер нагоняет ведущий, в `pg_stat_replication` показывается, сколько времени потребовалось для записи, сброса на диск и воспроизведения последних данных WAL, а не 0, как могли ожидать некоторые пользователи. Это соответствует задаче измерения задержек синхронной фиксации и видимости транзакций для недавно записанных транзакций. Чтобы меньше смущать пользователей, ожидающих видеть другую модель задержек, столбцы задержек сбрасываются в NULL после небольшой паузы в системе, которая воспроизвела все изменения и теперь простаивает. Системы мониторинга могут представлять это как отсутствующие данные, 0 или продолжать показывать последнее известное значение.

Таблица 27.6. Представление `pg_stat_wal_receiver`

Столбец	Тип	Описание
`pid`	`integer`	Идентификатор процесса WAL-приёмника
`status`	`text`	Состояние активности процесса WAL-приёмника
`receive_start_lsn`	`pg_lsn`	Первая позиция в журнале предзаписи в момент запуска приёмника WAL
`receive_start_tli`	`integer`	Первый номер линии времени в момент запуска приёмника WAL
`received_lsn`	`pg_lsn`	Последняя позиция в журнале предзаписи, уже полученная и сброшенная на диск; начальным значением этого поля будет первая позиция в журнале в момент запуска приёмника WAL
`received_tli`	`integer`	Номер линии времени последней позиции в журнале предзаписи, уже полученной и сброшенной на диск; начальным значением этого поля будет номер линии времени первой позиции в момент запуска приёмника WAL
`last_msg_send_time`	`timestamp with time zone`	Время отправки последнего сообщения, полученного от изначального передатчика WAL
`last_msg_receipt_time`	`timestamp with time zone`	Время поступления последнего сообщения, полученного от изначального передатчика WAL

Столбец	Тип	Описание
latest_end_lsn	pg_lsn	Последняя позиция в журнале предзаписи, сообщённая изначальному передатчику WAL
latest_end_time	timestamp with time zone	Время последней позиции в журнале предзаписи, сообщённой изначальному передатчику WAL
slot_name	text	Имя слота репликации, используемого этим приёмником WAL
conninfo	text	Строка подключения, используемая этим приёмником WAL (секретные поля в ней скрыты).

Представление pg_stat_wal_receiver будет иметь только одну строку со статистикой приёмника WAL от сервера, на котором работает приёмник.

Таблица 27.7. Представление pg_stat_subscription

Столбец	Тип	Описание
subid	oid	OID подписки
subname	text	Имя подписки
pid	integer	Идентификатор рабочего процесса, обслуживающего подписку
relid	Oid	OID отношения, которое синхронизирует рабочий процесс сейчас; null для основного процесса применения изменений
received_lsn	pg_lsn	Последняя позиция в журнале предзаписи (начальное значение этого поля — 0)
last_msg_send_time	timestamp with time zone	Время отправки последнего сообщения, полученного от изначального передатчика WAL
last_msg_receipt_time	timestamp with time zone	Время поступления последнего сообщения, полученного от изначального передатчика WAL
latest_end_lsn	pg_lsn	Последняя позиция в журнале предзаписи, сообщённая изначальному передатчику WAL
latest_end_time	timestamp with time zone	Время последней позиции в журнале предзаписи, сообщённой изначальному передатчику WAL

Представление pg_stat_subscription содержит по одной строке для подписки для основного рабочего процесса (с NULL в PID, если процесс не работает) и дополнительные строки для рабочих процессов, осуществляющих копирование начальных данных для таблиц в подписке.

Таблица 27.8. Представление pg_stat_ssl

Столбец	Тип	Описание
pid	integer	Идентификатор обслуживающего процесса или процесса, передающего WAL
ssl	boolean	True, если для этого подключения используется SSL
version	text	Версия SSL либо NULL, если SSL для этого подключения не используется
cipher	text	Имя применяемого шифра SSL либо NULL, если SSL для этого подключения не используется
bits	integer	Число бит в применяемом алгоритме шифрования либо NULL, если SSL для этого подключения не используется
compression	boolean	True, если применяется сжатие SSL, false в противном случае, либо NULL, если SSL для этого подключения не используется
clientdn	text	Поле DN (Distinguished Name, Уникальное имя) из используемого клиентского сертификата либо NULL, если клиент не передал сертификат или SSL для этого подключения не используется. Это значение усекается, если поле DN оказывается длиннее NAMEDATALEN символов (64 символа в стандартной сборке)

Представление pg_stat_ssl содержит по одной строке для каждого обслуживающего процесса и процесса, передающего WAL, и показывает статистику использования SSL для подключений. Его можно соединить с pg_stat_activity или pg_stat_replication по столбцу pid и получить дополнительные сведения о подключении.

Таблица 27.9. Представление pg_stat_archiver

Столбец	Тип	Описание
archived_count	bigint	Число файлов WAL, которые были успешно архивированы
last_archived_wal	text	Имя последнего файла WAL успешно архивированного
last_archived_time	timestamp with time zone	Время последней успешной архивации
failed_count	bigint	Число неудачных попыток архивации файлов WAL

Столбец	Тип	Описание
last_failed_wal	text	Имя файла WAL последней неудавшейся архивации
last_failed_time	timestamp with time zone	Время последней неудавшейся архивации
stats_reset	timestamp with time zone	Последнее время сброса этих статистических данных

В представлении pg_stat_archiver всегда будет одна строка, содержащая данные о текущем состоянии процесса архивации в кластере.

Таблица 27.10. Представление pg_stat_bgwriter

Столбец	Тип	Описание
checkpoints_timed	bigint	Количество запланированных контрольных точек, которые уже были выполнены
checkpoints_req	bigint	Количество запрошенных контрольных точек, которые уже были выполнены
checkpoint_write_time	double precision	Общее время, которое было затрачено на этап обработки контрольной точки, в котором файлы записываются на диск, в миллисекундах
checkpoint_sync_time	double precision	Общее время, которое было затрачено на этап обработки контрольной точки, в котором файлы синхронизируются с диском, в миллисекундах
buffers_checkpoint	bigint	Количество буферов, записанных при выполнении контрольных точек
buffers_clean	bigint	Количество буферов, записанных фоновым процессом записи
maxwritten_clean	bigint	Сколько раз фоновый процесс записи останавливал сброс грязных страниц на диск из-за того, что записал слишком много буферов
buffers_backend	bigint	Количество буферов, записанных самим серверным процессом
buffers_backend_fsync	bigint	Сколько раз серверному процессу пришлось выполнить fsync самостоятельно (обычно фоновый процесс записи сам обрабатывает эти вызовы, даже когда серверный процесс выполняет запись самостоятельно)
buffers_alloc	bigint	Количество выделенных буферов

Столбец	Тип	Описание
stats_reset	timestamp with time zone	Последнее время сброса этих статистических данных

В представлении pg_stat_bgwriter всегда будет только одна строка, в которой будут представлены общие данные по всему кластеру.

Таблица 27.11. Представление pg_stat_database

Столбец	Тип	Описание
datid	oid	OID базы данных
datname	name	Имя базы данных
numbackends	integer	Количество серверных процессов, в настоящее время подключённых к этой базе данных. Это единственный столбец в этом представлении, значение в котором отражает текущее состояние; все другие столбцы возвращают суммарные значения со времени последнего сброса статистики.
xact_commit	bigint	Количество зафиксированных транзакций в этой базе данных
xact_rollback	bigint	Количество транзакций в этой базе данных, для которых был выполнен откат транзакции
blks_read	bigint	Количество прочитанных дисковых блоков в этой базе данных
blks_hit	bigint	Сколько раз дисковые блоки обнаруживались в буферном кеше, так что чтение с диска не потребовалось (в значение входят только случаи обнаружения в буферном кеше Postgres Pro, а не в кеше файловой системы ОС)
tup_returned	bigint	Количество строк, возвращённое запросами в этой базе данных
tup_fetched	bigint	Количество строк, извлечённое запросами в этой базе данных
tup_inserted	bigint	Количество строк, вставленное запросами в этой базе данных
tup_updated	bigint	Количество строк, изменённое запросами в этой базе данных
tup_deleted	bigint	Количество строк, удалённое запросами в этой базе данных
conflicts	bigint	Количество запросов, отменённых из-за конфликта с восстановлением в этой

Столбец	Тип	Описание
		базе данных. (Конфликты происходят только на ведомых серверах; более подробно смотрите pg_stat_database_conflicts.)
temp_files	bigint	Количество временных файлов, созданных запросами в этой базе данных. Подсчитываются все временные файлы независимо от причины их создания (например, для сортировки или для хеширования) и независимо от установленного значения log_temp_files
temp_bytes	bigint	Общий объём данных, записанных во временные файлы запросами в этой базе данных. Учитываются все временные файлы, вне зависимости от того, по какой причине они созданы и вне зависимости от значения log_temp_files.
deadlocks	bigint	Количество взаимных блокировок, зафиксированное в этой базе данных
blk_read_time	double precision	Время, затраченное серверными процессами в этой базе данных, на чтение блоков из файлов данных, в миллисекундах
blk_write_time	double precision	Время, затраченное серверными процессами в этой базе данных, на запись блоков в файлы данных, в миллисекундах
stats_reset	timestamp with time zone	Последнее время сброса этих статистических данных

Представление pg_stat_database содержит одну строку со статистикой на каждую базу данных кластера.

Таблица 27.12. Представление pg_stat_database_conflicts

Столбец	Тип	Описание
datid	oid	OID базы данных
datname	name	Имя базы данных
confl_tablespace	bigint	Количество запросов в этой базе данных, отменённых из-за того, что табличные пространства были удалены
confl_lock	bigint	Количество запросов в этой базе данных, отменённых по

Столбец	Тип	Описание
		истечении времени ожидания блокировки
confl_snapshot	bigint	Количество запросов в этой базе данных, отменённых из-за устаревших снимков данных
confl_bufferpin	bigint	Количество запросов в этой базе данных, отменённых из-за прикреплённых страниц буфера
confl_deadlock	bigint	Количество запросов в этой базе данных, отменённых из-за взаимных блокировок

Представление pg_stat_database_conflicts для каждой базы данных будет содержать по одной строке со статистикой на уровне базы по отменам запросов, произошедшим вследствие конфликтов с процессами восстановления на ведомых серверах. В этом представлении будет содержаться только информация по ведомым серверам, поскольку на главных серверах конфликты не возникают.

Таблица 27.13. Представление pg_stat_all_tables

Столбец	Тип	Описание
relid	oid	OID таблицы
schemaname	name	Имя схемы, в которой расположена эта таблица
relname	name	Имя таблицы
seq_scan	bigint	Количество последовательных чтений, произведённых в этой таблице
seq_tup_read	bigint	Количество «живых» строк, прочитанных при последовательных чтениях
idx_scan	bigint	Количество сканирований по индексу, произведённых в этой таблице
idx_tup_fetch	bigint	Количество «живых» строк, отобранных при сканированиях по индексу
n_tup_ins	bigint	Количество вставленных строк
n_tup_upd	bigint	Количество изменённых строк (включая изменения по схеме HOT)
n_tup_del	bigint	Количество удалённых строк
n_tup_hot_upd	bigint	Количество строк, обновлённых в режиме HOT (т. е. без отдельного изменения индекса)
n_live_tup	bigint	Оценочное количество «живых» строк
n_dead_tup	bigint	Оценочное количество «мёртвых» строк
n_mod_since_analyze	bigint	Оценочное число строк, изменённых в этой таблице

Столбец	Тип	Описание
		с момента последнего сбора статистики
last_vacuum	timestamp with time zone	Время последней очистки этой таблицы вручную (VACUUM FULL не учитывается)
last_autovacuum	timestamp with time zone	Время последней очистки таблицы фоновым процессом автоочистки
last_analyze	timestamp with time zone	Время последнего выполнения сбора статистики для этой таблицы вручную
last_autoanalyze	timestamp with time zone	Время последнего выполнения сбора статистики для этой таблицы фоновым процессом автоочистки
vacuum_count	bigint	Сколько раз очистка этой таблицы была выполнена вручную (VACUUM FULL не учитывается)
autovacuum_count	bigint	Сколько раз очистка этой таблицы была выполнена фоновым процессом автоочистки
analyze_count	bigint	Сколько раз сбор статистики для этой таблицы был выполнен вручную
autoanalyze_count	bigint	Сколько раз сбор статистики для этой таблицы был выполнен фоновым процессом автоочистки

Представление pg_stat_all_tables будет содержать по одной строке на каждую таблицу в текущей базе данных (включая таблицы TOAST) со статистикой по обращениям к этой таблице. Представления pg_stat_user_tables и pg_stat_sys_tables содержат ту же самую информацию, но отфильтрованную так, чтобы показывать только пользовательские и системные таблицы соответственно.

Таблица 27.14. Представление pg_stat_all_indexes

Столбец	Тип	Описание
relid	oid	OID таблицы для индекса
indexrelid	oid	OID индекса
schemaname	name	Имя схемы, в которой расположен индекс
relname	name	Имя таблицы для индекса
indexrelname	name	Имя индекса
idx_scan	bigint	Количество произведённых сканирований по этому индексу
idx_tup_read	bigint	Количество элементов индекса, возвращённых при

Столбец	Тип	Описание
		сканированиях по этому индексу
idx_tup_fetch	bigint	Количество живых строк таблицы, отобранных при простых сканированиях по этому индексу

Представление pg_stat_all_indexes для каждого индекса в текущей базе данных будет содержать по одной строке со статистикой по обращениям к этому индексу. Представления pg_stat_user_indexes и pg_stat_sys_indexes содержат ту же самую информацию, но отфильтрованную так, чтобы показывать только пользовательские и системные индексы соответственно.

Индексы могут использоваться при простом сканировании по индексу, при сканировании «битовой карты» индекса и в работе оптимизатора. Результаты сканирования битовых карт разных индексов могут объединяться логическим умножением или сложением, поэтому когда применяются битовые карты, сложно связать выборки отдельных строк с определёнными индексами. Поэтому при сканировании битовых карт увеличиваются счётчики pg_stat_all_indexes.idx_tup_read для задействованных индексов и счётчик pg_stat_all_tables.idx_tup_fetch для каждой таблицы, а pg_stat_all_indexes.idx_tup_fetch не меняется. Оптимизатор тоже обращается к индексам для проверки переданных констант, значения которых оказываются вне диапазона, записанного в статистике оптимизатора, так как эта статистика может быть неактуальной.

> ## Примечание
>
> Значения счётчиков idx_tup_read и idx_tup_fetch могут различаться, даже если сканирование с использованием битовой карты не используется, поскольку idx_tup_read подсчитывает полученные из индекса элементы, а idx_tup_fetch — количество «живых» строк, выбранных из таблицы. Различие будет меньше, если «мёртвые» или ещё не зафиксированные строки будут извлекаться с использованием индекса или если для получения строк таблицы будет использоваться сканирование только по индексу.

Таблица 27.15. Представление pg_statio_all_tables

Столбец	Тип	Описание
relid	oid	OID таблицы
schemaname	name	Имя схемы, в которой расположена эта таблица
relname	name	Имя таблицы
heap_blks_read	bigint	Количество дисковых блоков, прочитанных из этой таблицы
heap_blks_hit	bigint	Число попаданий в буфер для этой таблицы
idx_blks_read	bigint	Количество дисковых блоков, прочитанных из всех индексов этой таблицы
idx_blks_hit	bigint	Число попаданий в буфер для всех индексов по этой таблице
toast_blks_read	bigint	Количество прочитанных дисковых блоков TOAST (если есть) для этой таблицы

Столбец	Тип	Описание
toast_blks_hit	bigint	Число попаданий в буфер в таблице TOAST для этой таблицы (если такие есть)
tidx_blks_read	bigint	Количество прочитанных дисковых блоков из индекса по TOAST (если есть) для этой таблицы
tidx_blks_hit	bigint	Число попаданий в буфер для индекса по TOAST (если есть) для этой таблицы

Представление pg_statio_all_tables для каждой таблицы (включая таблицы TOAST) в текущей базе данных будет содержать по одной строке со статистикой по операциям ввода/вывода для этой таблицы. Представления pg_statio_user_tables и pg_statio_sys_tables содержат ту же самую информацию, но отфильтрованную так, чтобы показывать только пользовательские или системные таблицы соответственно.

Таблица 27.16. Представление pg_statio_all_indexes

Столбец	Тип	Описание
relid	oid	OID таблицы для индекса
indexrelid	oid	OID индекса
schemaname	name	Имя схемы, в которой расположен индекс
relname	name	Имя таблицы для индекса
indexrelname	name	Имя индекса
idx_blks_read	bigint	Количество дисковых блоков, прочитанных из этого индекса
idx_blks_hit	bigint	Число попаданий в буфер для этого индекса

Представление pg_statio_all_indexes для каждого индекса в текущей базе данных будет содержать по одной строке со статистикой по операциям ввода/вывода для этого индекса. Представления pg_statio_user_indexes и pg_statio_sys_indexes содержат ту же самую информацию, но отфильтрованную так, чтобы показывать только пользовательские или системные индексы соответственно.

Таблица 27.17. Представление pg_statio_all_sequences

Столбец	Тип	Описание
relid	oid	OID последовательности
schemaname	name	Имя схемы, в которой расположена эта последовательность
relname	name	Имя последовательности
blks_read	bigint	Количество дисковых блоков, прочитанных из этой последовательности
blks_hit	bigint	Число попаданий в буфер в этой последовательности

Представление pg_statio_all_sequences для каждой последовательности в текущей базе данных будет содержать по одной строке со статистикой по операциям ввода/вывода для этой последовательности.

Таблица 27.18. Представление pg_stat_user_functions

Столбец	Тип	Описание
funcid	oid	OID функции
schemaname	name	Имя схемы, в которой расположена функция
funcname	name	Имя функции
calls	bigint	Сколько раз вызывалась функция
total_time	double precision	Общее время, потраченное на выполнение этой функции и всех других функций, вызванных ею, в миллисекундах
self_time	double precision	Общее время, потраченное на выполнение самой функции, без учёта других функций, которые были ею вызваны, в миллисекундах

Представление pg_stat_user_functions для каждой отслеживаемой функции будет содержать по одной строке со статистикой по выполнениям этой функции. Отслеживаемые функции определяются параметром track_functions.

27.2.3. Статистические функции

Статистическую информацию можно просматривать и другими способами. Для этого можно написать запросы, использующие те же функции доступа к статистике, что лежат в основе описанных выше стандартных представлений. За более подробной информацией, например, об именах этих функций, обратитесь к определениям этих стандартных представлений. (Например, в psql можно выполнить \d+ pg_stat_activity.) В качестве аргумента функции, предоставляющие доступ к статистике на уровне базы, принимают OID базы данных, по которой должна быть выдана информация. Функции, которые работают на уровне таблиц и индексов, принимают в качестве аргумента OID таблицы или индекса. Аргументом для функции, предоставляющей статистику на уровне функций, является OID функции. Обратите внимание, что с помощью этих функций можно получить информацию по таблицам, индексам и функциям исключительно в текущей базе данных.

Дополнительные функции, связанные со сбором статистики, перечислены в Таблице 27.19.

Таблица 27.19. Дополнительные статистические функции

Функция	Тип результата	Описание
pg_backend_pid()	integer	Идентификатор серверного процесса, выполняющего текущую сессию
pg_stat_get_activity (integer)	setof record	Возвращает запись с информацией о серверном процессе с заданным PID или по одной строке для каждого активного серверного процесса в системе, если был

Функция	Тип результата	Описание
		указан NULL. Возвращаемые поля являются подмножеством столбцов представления pg_stat_activity .
pg_stat_get_snapshot_timestamp()	timestamp with time zone	Возвращает время снимка текущей статистики
pg_stat_clear_snapshot()	void	Сбросить текущий снимок статистики
pg_stat_reset()	void	Сбросить в ноль все статистические счётчики в текущей базе данных (по умолчанию разрешено только суперпользователям, но право выполнения (EXECUTE) этой функции можно дать и другим)
pg_stat_reset_shared (text)	void	Сбросить в ноль некоторые статистические счётчики на уровне кластера, в зависимости от заданного аргумента (по умолчанию разрешено только суперпользователям, но право выполнения (EXECUTE) этой функции можно дать и другим). Вызов pg_stat_reset_shared('bgwriter') сбросит в ноль все счётчики, которые показываются в представлении pg_stat_bgwriter , а вызов pg_stat_reset_shared('archiver') — все счётчики в представлении pg_stat_archiver .
pg_stat_reset_single_table_counters (oid)	void	Сбросить в ноль статистику по отдельной таблице или индексу в текущей базе данных (по умолчанию разрешено только суперпользователям, но право выполнения (EXECUTE) этой функции можно дать и другим)
pg_stat_reset_single_function_counters (oid)	void	Сбросить в ноль статистику по отдельной функции в текущей базе данных (по умолчанию разрешено только суперпользователям, но право выполнения (EXECUTE) этой функции можно дать и другим)

Предупреждение

Функция pg_stat_reset() также сбрасывает счётчики, на основании которых процесс автоочистки принимает решение о запуске очистки или анализа. В результате сброса процесс автоочистки может не выполнить необходимые действия, что чревато

замусориванием таблиц, потерей актуальности статистики и тому подобными проблемами. Поэтому после сброса счётчиков рекомендуется выполнить ANALYZE на уровне базы.

Функция pg_stat_get_activity, на которой основано представление pg_stat_activity, возвращает набор строк, содержащих всю доступную информацию о каждом серверном процессе. Иногда более удобным оказывается получение только части этой информации. В таких случаях можно использовать набор более старых функций, дающих доступ к статистике на уровне серверных процессов; эти функции описаны в Таблице 27.20. Эти функции используют идентификатор серверного процесса, значение которого варьируется от единицы до числа активных в настоящий момент серверных процессов. Функция pg_stat_get_backend_idset генерирует по одной строке для каждого активного серверного процесса, что необходимо для вызова этих функций. Например, для того, чтобы отобразить значения PID и текущие запросы всех серверных процессов:

```
SELECT pg_stat_get_backend_pid(s.backendid) AS pid,
       pg_stat_get_backend_activity(s.backendid) AS query
    FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

Таблица 27.20. Статистические функции на уровне серверных процессов

Функция	Тип результата	Описание
pg_stat_get_backend_idset()	setof integer	Набор значений идентификаторов активных в настоящий момент серверных процессов (от 1 до числа активных серверных процессов)
pg_stat_get_backend_activity(integer)	text	Текст последнего запроса этого серверного процесса
pg_stat_get_backend_activity_start(integer)	timestamp with time zone	Время запуска последнего запроса
pg_stat_get_backend_client_addr(integer)	inet	IP-адрес клиента, подключённого к этому серверному процессу
pg_stat_get_backend_client_port(integer)	integer	Номер TCP-порта, который клиент использует для взаимодействия
pg_stat_get_backend_dbid(integer)	oid	OID базы данных, к которой подключён этот серверный процесс
pg_stat_get_backend_pid(integer)	integer	Идентификатор процесса этого серверного процесса
pg_stat_get_backend_start(integer)	timestamp with time zone	Время запуска этого процесса
pg_stat_get_backend_userid(integer)	oid	OID пользователя, подключённого к этому серверному процессу
pg_stat_get_backend_wait_event_type(integer)	text	Имя типа ожидаемого события, если обслуживающий процесс находится в состоянии ожидания, а в противном случае — NULL. За подробностями обратитесь к Таблице 27.4.

Функция	Тип результата	Описание
pg_stat_get_backend_wait_event(integer)	text	Имя ожидаемого события, если обслуживающий процесс находится в состоянии ожидания, а в противном случае — NULL. За подробностями обратитесь к Таблице 27.4.
pg_stat_get_backend_xact_start(integer)	timestamp with time zone	Время начала текущей транзакции

27.3. Просмотр информации о блокировках

Ещё одним удобным средством для отслеживания работы базы данных является системная таблица pg_locks. Она позволяет администратору базы просматривать информацию об имеющихся блокировках в менеджере блокировок. Например, это может использоваться для:

- просмотра всех имеющихся на данный момент блокировок, всех блокировок на отношения в определённой базе данных, всех блокировок на определённое отношение или всех блокировок, которые удерживает определённая сессия Postgres Pro.
- определения отношения в текущей базе данных с наибольшим количеством неразрешённых блокировок (оно может быть причиной конкуренции между клиентами базы данных).
- определения воздействия конкуренции за блокировку на производительность базы данных в целом, а так же то, как меняется конкуренция в зависимости от загруженности базы.

Более детально представление pg_locks описано в Разделе 49.73. Более подробную информацию о блокировках и управлению параллельным доступом в Postgres Pro можно получить в Главе 13.

27.4. Отслеживание выполнения

Postgres Pro имеет возможность отслеживать выполнение определённых команд. В настоящее время такое отслеживание поддерживает только команда VACUUM, но в будущем сфера его применения может быть расширена.

27.4.1. Отслеживание выполнения VACUUM

Во время выполнения VACUUM представление pg_stat_progress_vacuum будет содержать по одной строке для каждого обслуживающего процесса (включая рабочие процессы автоочистки), производящего очистку. Таблицы ниже показывают, какая информация будет отслеживаться, и рассказывают, как её интерпретировать. В настоящее время отслеживание выполнения не поддерживается для команды VACUUM FULL, так что процессы, выполняющие VACUUM FULL, не будут видны в этом представлении.

Таблица 27.21. Представление pg_stat_progress_vacuum

Столбец	Тип	Описание
pid	integer	Идентификатор (PID) обслуживающего процесса
datid	oid	OID базы данных, к которой подключён этот обслуживающий процесс.
datname	name	Имя базы данных, к которой подключён этот обслуживающий процесс.
relid	oid	OID очищаемой таблицы.
phase	text	Текущая фаза очистки. См. Таблицу 27.22.

Столбец	Тип	Описание
heap_blks_total	bigint	Общее число блоков кучи в таблице. Это число отражает состояние в начале сканирования; блоки, добавленные позже, не будут (и не должны) обрабатываться текущей командой VACUUM.
heap_blks_scanned	bigint	Число просканированных блоков кучи. Так как для оптимизации сканирования применяется карта видимости, некоторые блоки могут пропускаться без осмотра; пропущенные блоки входят в это общее число, так что по завершении очистки это число станет равно heap_blks_total . Этот счётчик увеличивается только в фазе scanning heap.
heap_blks_vacuumed	bigint	Число очищенных блоков кучи. Если в таблице нет индексов, этот счётчик увеличивается только в фазе vacuuming heap (очистка кучи). Блоки, не содержащие «мёртвых» кортежей, при этом пропускаются, так что этот счётчик иногда может увеличиваться резкими рывками.
index_vacuum_count	bigint	Количество завершённых циклов очистки индекса.
max_dead_tuples	bigint	Число «мёртвых» кортежей, которое мы можем сохранить, прежде чем потребуется выполнить цикл очистки индекса, в зависимости от maintenance_work_mem.
num_dead_tuples	bigint	Число «мёртвых» кортежей, собранных со времени последнего цикла очистки индекса.

Таблица 27.22. Фазы VACUUM

Фаза	Описание
initializing	Инициализация — VACUUM готовится начать сканирование кучи. Эта фаза должна быть очень быстрой.
scanning heap	Сканирование кучи — VACUUM в настоящее время сканирует кучу. При этом будет очищена и, если требуется, дефрагментирована каждая страница, а возможно, также будет произведена

Фаза	Описание
	заморозка. Отслеживать процесс сканирования можно, следя за содержимым столбца heap_blks_scanned .
vacuuming indexes	Очистка индексов — VACUUM в настоящее время очищает индексы. Если у таблицы есть какие-либо индексы, эта фаза будет наблюдаться минимум единожды в процессе очистки, после того, как куча будет просканирована полностью. Она может повторяться несколько раз в процессе очистки, если объёма maintenance_work_mem (или, в случае автоочистки, autovacuum_work_mem, если он задан) оказывается недостаточно для сохранения всех найденных «мёртвых» кортежей.
vacuuming heap	Очистка кучи — VACUUM в настоящее время очищает кучу. Очистка кучи отличается от сканирования, так как она происходит после каждой операции очистки индексов. Если heap_blks_scanned меньше чем heap_blks_total , система вернётся к сканированию кучи после завершения этой фазы; в противном случае она начнёт уборку индексов.
cleaning up indexes	Уборка индексов — VACUUM в настоящее время производит уборку в индексах. Это происходит после завершения полного сканирования кучи и очистки индексов и кучи.
truncating heap	Усечение кучи — VACUUM в настоящее время усекает кучу, чтобы возвратить операционной системе объём пустых страниц в конце отношения. Это происходит после уборки индексов.
performing final cleanup	Выполнение окончательной очистки — VACUUM выполняет окончательную очистку. На этой стадии VACUUM очищает карту свободного пространства, обновляет статистику в pg_class и передаёт статистику сборщику статистики, После этой фазы VACUUM завершит свою работу.

27.5. Динамическая трассировка

Postgres Pro позволяет выполнять динамическую трассировку сервера базы данных. Имеющиеся возможности позволяют вызывать внешнюю утилиту в определённых точках кода и таким образом отслеживать его выполнение.

Несколько подобных точек сбора метрик, или точек трассировки, уже встроено в исходный код. Предполагается, что эти точки будут использоваться разработчиками и администраторами базы данных. По умолчанию точки трассировки не входят в сборку Postgres Pro; пользователь должен явно указать конфигурационному скрипту необходимость включения этих макросов.

В настоящее время поддерживается только утилита *DTrace*, которая доступна для Solaris, macOS, FreeBSD, NetBSD и Oracle Linux. Проект *SystemTap* для Linux представляет собой эквивалент DTrace и также может быть использован. Теоретически возможна поддержка и других утилит динамической трассировки, для этого необходимо изменить определения для макроса в src/include/utils/probes.h.

27.5.1. Компиляция для включения динамической трассировки

По умолчанию точки трассировки недоступны, поэтому в конфигурационном скрипте Postgres Pro требуется явно указать необходимость их подключения. Для поддержки утилиты DTrace укажите `--enable-dtrace` при конфигурировании.

27.5.2. Встроенные точки трассировки

В исходный код входит несколько стандартных точек трассировки, которые представлены в Таблице 27.23; в Таблице 27.24 показаны типы данных, которые используются для этих точек. Конечно, для более детального отслеживания работы Postgres Pro можно добавлять и другие точки трассировки.

Таблица 27.23. Встроенные точки трассировки DTrace

Имя	Параметры	Описание
`transaction-start`	`(LocalTransactionId)`	Срабатывает в начале новой транзакции. arg0 задаёт идентификатор транзакции.
`transaction-commit`	`(LocalTransactionId)`	Срабатывает при успешном завершении транзакции. arg0 задаёт идентификатор транзакции.
`transaction-abort`	`(LocalTransactionId)`	Срабатывает, когда транзакция завершается с ошибкой. arg0 задаёт идентификатор транзакции.
`query-start`	`(const char *)`	Срабатывает, когда начинается обработка запроса. arg0 задаёт текст запроса.
`query-done`	`(const char *)`	Срабатывает по завершении обработки запроса. arg0 задаёт текст запроса.
`query-parse-start`	`(const char *)`	Срабатывает, когда начинается разбор запроса. arg0 задаёт текст запроса.
`query-parse-done`	`(const char *)`	Срабатывает по завершении разбора (parsing) запроса. arg0 задаёт текст запроса.
`query-rewrite-start`	`(const char *)`	Срабатывает, когда начинается модификация запроса. arg0 задаёт текст запроса.
`query-rewrite-done`	`(const char *)`	Срабатывает по завершении модификации запроса. arg0 задаёт текст запроса.
`query-plan-start`	`()`	Срабатывает, когда начинает работать планировщик выполнения запроса.
`query-plan-done`	`()`	Срабатывает по завершении работы планировщика запроса.
`query-execute-start`	`()`	Срабатывает, когда начинается выполнение запроса.
`query-execute-done`	`()`	Срабатывает по завершении выполнения запроса.

Имя	Параметры	Описание
statement-status	(const char *)	Срабатывает каждый раз, когда серверный процесс обновляет свой статус в pg_stat_activity.status. arg0 задаёт новую строку состояния.
checkpoint-start	(int)	Срабатывает в начале контрольной точки. arg0 содержит битовые флаги, с помощью которых задаются разные типы контрольных точек, такие как shutdown, immediate или force.
checkpoint-done	(int, int, int, int, int)	Срабатывает по завершении контрольной точки. (Перечисленные далее точки трассировки срабатывают последовательно при обработке контрольной точки.) arg0 задаёт число записанных буферов. arg1 — общее число буферов. arg2, arg3 и arg4 задают число файлов WAL, которые были добавлены, удалены или переработаны, соответственно.
clog-checkpoint-start	(bool)	Срабатывает, когда начинается запись контрольной точки в CLOG. arg0 = true для обычных контрольных точек и false для контрольных точек типа shutdown.
clog-checkpoint-done	(bool)	Срабатывает по завершении записи контрольной точки в CLOG. Значение arg0 задаётся аналогично значению для clog-checkpoint-start.
subtrans-checkpoint-start	(bool)	Срабатывает, когда начинается запись контрольной точки в SUBTRANS. arg0 = true для обычных контрольных точек и false для контрольных точек типа shutdown.
subtrans-checkpoint-done	(bool)	Срабатывает по завершении записи контрольной точки в SUBTRANS. Значение arg0 задаётся аналогично значению для subtrans-checkpoint-start.
multixact-checkpoint-start	(bool)	Срабатывает, когда начинается запись контрольной точки в MultiXact. arg0 = true для обычных контрольных точек и

Имя	Параметры	Описание
		false для контрольных точек типа shutdown.
multixact-checkpoint-done	(bool)	Срабатывает по завершении записи контрольной точки в MultiXact. Значение arg0 задаётся аналогично значению для multixact-checkpoint-start.
buffer-checkpoint-start	(int)	Срабатывает, когда начинается запись буферов контрольной точки. arg0 содержит битовые флаги, с помощью которых задаются разные типы контрольных точек, такие как shutdown, immediate или force.
buffer-sync-start	(int, int)	Срабатывает во время контрольной точки, когда начинается запись грязных буферов (после нахождения буферов, которые должны быть записаны). arg0 задаёт общее число буферов. arg1 задаёт число буферов, которые в настоящий момент являются грязными и должны быть записаны.
buffer-sync-written	(int)	Срабатывает после записи каждого буфера при выполнении контрольной точки. arg0 задаёт идентификатор буфера.
buffer-sync-done	(int, int, int)	Срабатывает после записи всех грязных буферов. arg0 задаёт общее число буферов. arg1 задаёт число буферов, которые фактически были записаны процессом выполнения контрольной точки. arg2 задаёт число буферов, которое должно было быть записано (arg1 из buffer-sync-start); разные значения говорят о том, что во время выполнения этой контрольной точки буферы сбрасывались другими процессами.
buffer-checkpoint-sync-start	()	Срабатывает после записи грязных буферов в ядро и до начала формирования запросов fsync.
buffer-checkpoint-done	()	Срабатывает по завершении синхронизации буферов с диском.

Имя	Параметры	Описание
twophase-checkpoint-start	()	Срабатывает, когда начинается двухфазный этап выполнения контрольной точки.
twophase-checkpoint-done	()	Срабатывает по завершении двухфазного этапа выполнения контрольной точки.
buffer-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool)	Срабатывает, когда начинается чтение из буфера. arg0 и arg1 содержат номер слоя и блока этой страницы (arg1 будет иметь значение -1, если выполняется запрос на расширение места для таблицы). arg2, arg3 и arg4 содержат OID-ы табличного пространства, базы данных и отношения, которые однозначно идентифицируют отношение. arg5 для локального буфера задаёт идентификатор серверного процесса, создавшего временное отношение, или InvalidBackendId (-1) — для разделяемого буфера. arg6 = true для запросов на расширение места для таблицы, false — в случае обычного чтения.
buffer-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool, bool)	Срабатывает по завершении чтения буфера. arg0 и arg1 содержат номер слоя и номер блока этой страницы (arg1 будет содержать номер только что добавленного блока, если выполняется запрос на расширение места для таблицы). arg2, arg3 и arg4 содержат OID-ы табличного пространства, базы данных и отношения, которые однозначно идентифицируют отношение. arg5 для локального буфера задаёт идентификатор серверного процесса, создавшего временное отношение, или InvalidBackendId (-1) — для разделяемого буфера. arg6 = true для запросов на расширение места для таблицы, false — в случае обычного чтения. arg7 = true, если буфер был обнаружен в пуле, false — если нет.

Имя	Параметры	Описание
buffer-flush-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Срабатывает перед формированием любого запроса на запись в разделяемый буфер. arg0 и arg1 содержат номер слоя и номер блока этой страницы. arg2, arg3 и arg4 содержат OID-ы табличного пространства, базы данных и отношения, которые однозначно идентифицируют отношение.
buffer-flush-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Срабатывает по завершении запроса на запись. (Учтите, что это отражает только момент передачи данных в ядро; обычно на диск они ещё не записаны.) Аргументы аналогичны buffer-flush-start.
buffer-write-dirty-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Срабатывает, когда серверный процесс начинает запись грязного буфера. (Частое повторение такой пробы означает, что значение shared_buffers слишком мало или что необходимо откорректировать управляющие параметры процесса фоновой записи.) arg0 и arg1 содержат номер слоя и блока этой страницы. arg2, arg3 и arg4 содержат OID-ы табличного пространства, базы данных и отношения, которые однозначно идентифицируют отношение.
buffer-write-dirty-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Срабатывает по завершении записи грязного буфера. Аргументы аналогичны buffer-write-dirty-start.
wal-buffer-write-dirty-start	()	Срабатывает, когда серверный процесс начинает запись грязного WAL буфера из-за того, что свободные WAL буферы закончились. (Частое повторение такой ситуации означает, что значение wal_buffers слишком мало.)
wal-buffer-write-dirty-done	()	Срабатывает по завершении записи грязного WAL буфера.
wal-insert	(unsigned char, unsigned char)	Срабатывает при добавлении записи в WAL. arg0 задаёт идентификатор менеджера ресурсов (rmid) для этой

Имя	Параметры	Описание
		записи. arg1 задаёт информационные флаги.
wal-switch	()	Срабатывает при запросе на переключение сегмента WAL.
smgr-md-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	Срабатывает, когда начинается чтение блока из отношения. arg0 и arg1 содержат номер слоя и номер блока этой страницы. arg2, arg3 и arg4 содержат OID-ы табличного пространства, базы данных и отношения, которые однозначно идентифицируют отношение. arg5 для локального буфера задаёт идентификатор серверного процесса, создавшего временное отношение, или InvalidBackendId (-1) для разделяемого буфера.
smgr-md-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	Срабатывает по завершении чтения блока. arg0 и arg1 содержат номер слоя и номер блока страницы. arg2, arg3 и arg4 содержат OID-ы табличного пространства, базы данных и отношения, которые однозначно идентифицируют отношение. arg5 для локального буфера задаёт идентификатор серверного процесса, создавшего временное отношение, или InvalidBackendId (-1) — для разделяемого буфера. arg6 задаёт количество фактически прочитанных байтов, тогда как arg7 задаёт количество запрошенных байтов (различия говорят о наличии проблемы).
smgr-md-write-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	Срабатывает, когда начинается запись блока в отношение. arg0 и arg1 содержат номер слоя и номер блока этой страницы. arg2, arg3 и arg4 содержат OID-ы табличного пространства, базы данных и отношения, которые однозначно идентифицируют отношение. arg5 для локального буфера задаёт идентификатор серверного процесса, создавшего временное отношение, или

Имя	Параметры	Описание
		`InvalidBackendId` (-1) — для разделяемого буфера.
`smgr-md-write-done`	`(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)`	Срабатывает по завершении записи блока. arg0 и arg1 содержат номер слоя и номер блока этой страницы. arg2, arg3 и arg4 содержат OID-ы табличного пространства, базы данных и отношения, которые однозначно идентифицируют отношение. arg5 для локального буфера задаёт идентификатор серверного процесса, создавшего временное отношение, или `InvalidBackendId` (-1) — для разделяемого буфера. arg6 задаёт количество фактически записанных байтов, тогда как arg7 задаёт количество запрошенных байтов (различия говорят о наличии проблемы).
`sort-start`	`(int, bool, int, int, bool)`	Срабатывает, когда начинается операция сортировки. arg0 задаёт сортировку таблицы, индекса или элемента данных. arg1 = true, если данные ожидаются уникальными. arg2 задаёт число ключевых столбцов. arg3 задаёт объём доступной рабочей памяти в килобайтах. arg4 = true, если требуется произвольный доступ к результату сортировки.
`sort-done`	`(bool, long)`	Срабатывает по завершении сортировки. arg0 = true для внешней сортировки, false — для внутренней сортировки. arg1 задаёт число дисковых блоков, использованных для внешней сортировки, или объём памяти, использованной для внутренней сортировки, в килобайтах.
`lwlock-acquire`	`(char *, LWLockMode)`	Срабатывает, когда выдаётся блокировка LWLock. В arg0 передаётся транш блокировки, в arg1 запрошенный режим блокировки (исключительная или разделяемая).
`lwlock-release`	`(char *)`	Срабатывает, когда блокировка LWLock освобождается (но учтите, что никакие ждущие процессы ещё не пробуждены).

Имя	Параметры	Описание
		В arg0 передаётся транш блокировки.
lwlock-wait-start	(char *, LWLockMode)	Срабатывает, когда блокировка LWLock не доступна моментально, и серверный процесс начал ожидать её доступности. В arg0 передаётся транш блокировки, в arg1 запрошенный режим блокировки (исключительная или разделяемая).
lwlock-wait-done	(char *, LWLockMode)	Срабатывает, когда серверный процесс прекращает ожидание блокировки LWLock (но саму блокировку он ещё не получил). В arg0 передаётся транш блокировки, в arg1 запрошенный режим блокировки (исключительная или разделяемая).
lwlock-condacquire	(char *, LWLockMode)	Срабатывает, когда блокировка LWLock была успешно получена процессом, запросившим её в режиме без ожидания. В arg0 передаётся транш блокировки, в arg1 запрошенный режим блокировки (исключительная или разделяемая).
lwlock-condacquire-fail	(char *, LWLockMode)	Срабатывает, когда блокировка LWLock не была успешно получена процессом, запросившим её в режиме без ожидания. В arg0 передаётся транш блокировки, в arg1 запрошенный режим блокировки (исключительная или разделяемая).
lock-wait-start	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	Срабатывает, когда запрос на тяжёлую блокировку (блокировку lmgr) переходит в состояние ожидания, поскольку блокировка недоступна. Аргументы с arg0 до arg3 задают атрибуты, идентифицирующие объект, на который накладывается блокировка. arg4 задаёт тип объекта, на который накладывается блокировка. arg5 задаёт тип запрошенной блокировки.
lock-wait-done	(unsigned int, unsigned int, unsigned int, unsigned	Срабатывает, когда запрос на тяжёлую блокировку (блокировку lmgr) выходит

Имя	Параметры	Описание
	int, unsigned int, LOCKMODE)	из состояния ожидания (т. е. получает блокировку). Аргументы аналогичны lock-wait-start.
deadlock-found	()	Срабатывает, когда детектор взаимных блокировок обнаруживает такую взаимную блокировку

Таблица 27.24. Предопределённые типы, используемые в параметрах точек трассировки

Тип	Определение
LocalTransactionId	unsigned int
LWLockMode	int
LOCKMODE	int
BlockNumber	unsigned int
Oid	unsigned int
ForkNumber	int
bool	char

27.5.3. Использование точек трассировки

В приведённом ниже примере показан скрипт DTrace для анализа числа транзакций в системе, который можно использовать в качестве альтернативы созданию снимка данных pg_stat_database до и после выполнения теста производительности:

```
#!/usr/sbin/dtrace -qs

postgresql$1:::transaction-start
{
      @start["Start"] = count();
      self->ts  = timestamp;
}

postgresql$1:::transaction-abort
{
      @abort["Abort"] = count();
}

postgresql$1:::transaction-commit
/self->ts/
{
      @commit["Commit"] = count();
      @time["Total time (ns)"] = sum(timestamp - self->ts);
      self->ts=0;
}
```

При выполнении этот D-скрипт возвращает результат вида:

```
# ./txn_count.d `pgrep -n postgres` or ./txn_count.d <PID>
^C

Start                                          71
Commit                                         70
Total time (ns)                          2312105013
```

> **Примечание**
>
> SystemTap использует отличную от DTrace нотацию для скриптов трассировки, хотя лежащие в их основе точки трассировки совместимы. Стоит отметить, что на момент написания этой главы в скриптах SystemTap имена точек трассировки должны обрамляться двойными подчёркиваниями, а не дефисами. Ожидается, что эта проблема будет решена в следующих версиях SystemTap.

Необходимо помнить, что скрипты DTrace должны быть аккуратно написаны и отлажены, в противном случае собранная трассировочная информация может оказаться бессмысленной. В большинстве случаев причиной обнаруженных проблем является инструментарий, а не сама система. Отправляя на рассмотрение данные, полученные с использованием динамической трассировки, обязательно прилагайте скрипт, с помощью которого они были получены, для того чтобы его также проверить и обсудить.

27.5.4. Задание новых точек трассировки

Новые точки трассировки разработчик может определить в любом участке кода, однако это потребует перекомпиляции. Ниже приведены шаги, необходимые для добавления новых точек трассировки:

1. Определить имена точек трассировки и данные, которые будут доступны в этих точках
2. Добавить описание точек трассировки в `src/backend/utils/probes.d`
3. Включить `pg_trace.h`, если его ещё не использовали в модуле (модулях), содержащих точки трассировки, и вставить `TRACE_POSTGRESQL` отладочные макросы в нужные места исходного кода
4. Перекомпилировать и убедиться в доступности новых точек трассировки

Пример: Вот пример того, как можно добавить точку для трассировки всех новых транзакций по их идентификатору.

1. Устанавливаем, что проба будет называться `transaction-start` и принимать параметр типа `LocalTransactionId`
2. Добавляем определение пробы в `src/backend/utils/probes.d`:

   ```
   probe transaction__start(LocalTransactionId);
   ```

   Обратите внимание на использование двойного подчёркивания в имени пробы. В скрипте DTrace, использующем эту точку, двойное подчёркивание нужно будет заменить дефисом, поэтому в документации для пользователей имя этой пробы — `transaction-start`.
3. Во время компиляции `transaction__start` преобразуется в макрос `TRACE_POSTGRESQL_TRANSACTION_START` (обратите внимание, что здесь используется одинарное подчёркивание), который доступен в результате включения `pg_trace.h`. Добавим вызов макроса в требуемую точку исходного кода. В данном случае это будет выглядеть приблизительно так:

   ```
   TRACE_POSTGRESQL_TRANSACTION_START(vxid.localTransactionId);
   ```
4. После перекомпиляции и запуска нового бинарного файла используйте следующую команду DTrace, чтобы проверить доступность только что добавленной пробы. Должен получиться результат, подобный этому:

   ```
   # dtrace -ln transaction-start
      ID    PROVIDER          MODULE           FUNCTION NAME
   18705 postgresql49878     postgres     StartTransactionCommand transaction-start
   18755 postgresql49877     postgres     StartTransactionCommand transaction-start
   18805 postgresql49876     postgres     StartTransactionCommand transaction-start
   18855 postgresql49875     postgres     StartTransactionCommand transaction-start
   ```

```
 18986 postgresql49873     postgres     StartTransactionCommand transaction-start
```

При добавлении макросов трассировки в код, написанный на языке C, необходимо позаботиться о следующем:

- Нужно убедиться, что типы данных, определённые в параметрах пробы, совпадают с типами данных переменных, которые используются в макросе. В противном случае компиляция завершится с ошибками.
- В большинстве платформ в случае, если Postgres Pro собран с указанием `--enable-dtrace`, то аргументы макроса трассировки вычисляются каждый раз, когда макрос получает управление, *даже если трассировка не выполняется*. Об этом не стоит беспокоиться, если вы просто возвращаете значения небольшого числа локальных переменных. Однако избегайте использования ресурсоёмких вызовов функций в аргументах. Если это необходимо, то постарайтесь защитить макрос проверкой, которая будет определять, действительно ли включена трассировка:

  ```
  if (TRACE_POSTGRESQL_TRANSACTION_START_ENABLED())
      TRACE_POSTGRESQL_TRANSACTION_START(some_function(...));
  ```

  Каждый макрос трассировки имеет соответствующий макрос `ENABLED`.

Глава 28. Мониторинг использования диска

В данной главе обсуждается как отслеживать использование дискового пространства в СУБД Postgres Pro.

28.1. Определение использования диска

Для каждой таблицы создаётся первичный дисковый файл кучи (heap), в котором хранится большая часть данных. Если в таблице есть столбцы с потенциально большими значениями, то также может быть и ассоциированный с этой таблицей файл TOAST, который используется для хранения слишком больших значений, не умещающихся в главной таблице (см. Раздел 63.2). На каждую таблицу TOAST, если она существует, будет существовать один индекс. Также там могут быть и индексы, ассоциированные с базовой таблицей. Каждая таблица и индекс хранятся в отдельном дисковом файле — возможно более чем в одном файле, если размер этого файла превышает один гигабайт. Преобразования имён для этих файлов описываются в Разделе 63.1.

Вы можете осуществлять мониторинг дискового пространства тремя способами: используя SQL-функции, перечисленные в Таблице 9.84, используя модуль oid2name или просматривая системные каталоги вручную. Вышеупомянутые SQL функции являются наиболее простыми и рекомендуются для использования. В продолжении этого раздела рассказывается, как просматривать системные каталоги.

Используя psql после недавнего применения команд VACUUM или ANALYZE, вы можете выполнить такой запрос, чтобы увидеть сколько дискового пространства использует какая-либо таблица:

```
SELECT pg_relation_filepath(oid), relpages FROM pg_class WHERE relname = 'customer';

 pg_relation_filepath | relpages
----------------------+----------
 base/16384/16806     |       60
(1 row)
```

Каждая страница обычно равна 8kb. (Помните, что `relpages` обновляется только командами `VACUUM`, `ANALYZE`, и несколькими командами DDL, такими как `CREATE INDEX`). Путь к файлу представляет интерес, если вы хотите проанализировать непосредственно файл на диске.

Чтобы посмотреть пространство, используемое таблицами TOAST, используйте следующий запрос:

```
SELECT relname, relpages
FROM pg_class,
     (SELECT reltoastrelid
      FROM pg_class
      WHERE relname = 'customer') AS ss
WHERE oid = ss.reltoastrelid OR
      oid = (SELECT indexrelid
             FROM pg_index
             WHERE indrelid = ss.reltoastrelid)
ORDER BY relname;

       relname        | relpages
----------------------+----------
 pg_toast_16806       |        0
 pg_toast_16806_index |        1
```

Вы можете легко посмотреть размеры индексов:

```
SELECT c2.relname, c2.relpages
FROM pg_class c, pg_class c2, pg_index i
WHERE c.relname = 'customer' AND
      c.oid = i.indrelid AND
```

```
      c2.oid = i.indexrelid
ORDER BY c2.relname;

       relname        | relpages
----------------------+----------
 customer_id_indexdex |       26
```

Также легко найти самые большие таблицы и индексы, используя эту информацию:

```
SELECT relname, relpages
FROM pg_class
ORDER BY relpages DESC;

       relname        | relpages
----------------------+----------
 bigtable             |     3290
 customer             |     3144
```

28.2. Ошибка переполнения диска

Основная цель мониторинга дисков для администратора БД — предотвратить их переполнение. Если переполнится диск с данными, повреждения данных не произойдёт, но могут не выполняться полезные действия. Если же переполнится диск, содержащий файлы WAL, это может привести к аварийному сбою сервера с последующим его отключением.

Если вы не можете освободить дополнительное пространство на диске, удалив какие-либо другие файлы, то можно перенести часть файлов базы данных на другие файловые системы, с помощью создания табличных пространств. Подробности об этом смотрите в Раздел 21.6.

> **Подсказка**
>
> Некоторые файловые системы плохо работают, когда они почти или совсем заполнены, так что не ждите пока диск заполнится полностью, чтобы выполнить необходимые действия.

Если ваша система поддерживает дисковые квоты для пользователей, вы должны позаботиться о квоте, выделенной для пользователя, от имени которого запускается сервер СУБД. Если эта квота будет исчерпана, последствия будут столь же негативными, как если просто закончится свободное место на диске.

Глава 29. Надёжность и журнал предзаписи

В данной главе рассказывается, как для обеспечения эффективной и надёжной работы используется журнал предзаписи.

29.1. Надёжность

Надёжность — это важное свойство любой серьёзной СУБД и Postgres Pro делает всё возможное, чтобы гарантировать надёжность своего функционирования. Один из аспектов надёжности состоит в том, что все данные записываются с помощью подтверждённых транзакций, которые сохраняются в энергонезависимой области, которая защищена от потери питания, сбоев операционной системы и аппаратных отказов (разумеется, за исключением отказа самой энергонезависимой области). Успешная запись данных в постоянное место хранения (диск или эквивалентный носитель) обычно всё, что требуется. Фактически, даже если компьютер полностью вышел из строя, если диски выжили, то они могут быть переставлены в другой похожий компьютер и все подтверждённые транзакции останутся неповреждёнными.

Хотя периодическая запись данных на пластины диска может выглядеть как простая операция, это не так, потому что диски значительно медленнее, чем оперативная память и процессор, а также потому что между оперативной памятью и пластинами диска есть некоторые механизмы кеширования. Во-первых, есть буферный кеш операционной системы, который кеширует частые запросы к блокам диска и комбинирует запись на диск. К счастью, все операционные системы предоставляют приложениям способ принудительной записи из буферного кеша на диск и Postgres Pro использует эту возможность. (Смотрите параметр wal_sync_method который отвечает за то как это делается.)

Далее, кеширование также может осуществляться контроллером диска; в особенности это касается RAID-контроллеров. В некоторых случаях это кеширование работает в режиме *сквозной записи*, что означает, что запись осуществляется на диск как только приходят данные. В других случаях, возможна работа в режиме *отложенной записи*, что означает, что запись осуществляется некоторое время спустя. Такой режим кеширования может создавать риск для надёжности, потому что память контроллера диска непостоянна и будет потеряна в случае потери питания. Лучшие контроллеры имеют так называемую *батарею резервного питания* (Battery-Backup Unit, BBU), которая сохраняет кеш контроллера на батарее, если пропадёт системное питание. После возобновления питания, данные, оставшиеся в кеше контроллера, будут записаны на диски.

И наконец, многие диски имеют внутренний кеш. На каких-то дисках он работает в режиме сквозной записи, на других — в режиме отложенной записи. В последнем случае с кешем диска связаны те же риски потери данных, что и с кешем контроллера дисков. В частности, кеш отложенной записи, сбрасывающийся при потере питания, часто имеют диски IDE и SATA потребительского класса. Также зависимый от питания кеш отложенной записи имеют многие SSD-накопители.

Обычно, такое кеширование можно выключить; однако то, как это делается, различается для операционной системы и для типа диска:

- В Linux параметры дисков IDE и SATA могут быть получены с помощью команды `hdparm -I`; кеширование записи включено, если за строкой `Write cache` следует `*`. Для выключения кеширования записи может быть использована команда `hdparm -W 0`. Параметры SCSI-дисков могут быть получены с помощью утилиты *sdparm*. Используйте `sdparm --get=WCE`, чтобы проверить, включено ли кеширование записи, и `sdparm --clear=WCE`, чтобы выключить его.
- Во FreeBSD параметры IDE-дисков могут быть получены с помощью команды `atacontrol`, а кеширование записи выключается при помощи установки параметра `hw.ata.wc=0` в файле `/boot/loader.conf`; Для SCSI-дисков параметры могут быть получены, используя команду `camcontrol identify`, а кеширование записи изменяется при помощи утилиты `sdparm`.

- В Solaris кешированием записи на диск управляет команда `format -e`. (Использование файловой системы Solaris ZFS, при включённом кешировании записи на диск, является безопасным, потому что она использует собственные команды сброса кеша на диск.)
- В Windows, если параметр `wal_sync_method` установлен в `open_datasync` (по умолчанию), кеширование записи на диск может быть выключено снятием галочки `My Computer\Open\`*`disk drive`*`\Properties\Hardware\Properties\Policies\Enable write caching on the disk`. В качестве альтернативы, можно установить параметр `wal_sync_method` в значение `fsync` или `fsync_writethrough`, что предотвращает кеширование записи.
- В macOS кеширование записи можно отключить, установив для параметра `wal_sync_method` значение `fsync_writethrough`.

Новые модели SATA-дисков (которые соответствуют стандарту ATAPI-6 или более позднему) предлагают команду сброса кеша на диск (`FLUSH CACHE EXT`), а SCSI-диски уже давно поддерживают похожую команду `SYNCHRONIZE CACHE`. Эти команды недоступны из Postgres Pro напрямую, но некоторые файловые системы (например, ZFS, ext4), могут использовать их для сброса данных из кеша на пластины диска при включённом режиме кеша сквозной записи. К сожалению, такие файловые системы ведут себя неоптимально при комбинировании с батареей резервного питания (BBU) дискового контроллера. В таких случаях, команда синхронизации принуждает сохранять все данные на диск из кеша контроллера, сводя преимущество BBU к нулю. Вы можете запустить модуль pg_test_fsync, чтобы увидеть, что вы попали в эту ситуацию. Если это так, преимущества производительности BBU могут быть восстановлены с помощью выключения барьеров записи для файловой системы или переконфигурирования контроллера диска, если это возможно. Если барьеры записи выключены, убедитесь, что батарея годная; при отказе батареи может произойти потеря данных. Есть надежда, что разработчики файловых систем и контроллеров дисков, в конце концов, устранят это неоптимальное поведение.

Когда операционная система отправляет запрос на запись к аппаратному обеспечению для хранения данных, она мало что может сделать, чтобы убедиться, что данные действительно сохранены в какой-либо энергонезависимой области. Скорее, это является зоной ответственности администратора, убедиться в целостности данных на всех компонентах хранения. Избегайте дисковых контроллеров, которые не имеют батарей резервного питания для кеширования записи. На уровне диска, запретите режим отложенной записи, если диск не может гарантировать, что данные будут записаны перед выключением. Если вы используете SSD, знайте, что многие из них по умолчанию не выполняют команды сброса кеша на диск. Вы можете протестировать надёжность поведения подсистемы ввода/вывода, используя *diskchecker.pl*.

Другой риск потери данных состоит в самой записи на пластины диска. Пластины диска разделяются на секторы, обычно по 512 байт каждый. Каждая операция физического чтения или записи обрабатывает целый сектор. Когда дисковый накопитель получает запрос на запись, он может соответствовать нескольким секторам по 512 байт (Postgres Pro обычно за один раз записывает 8192 байта или 16 секторов) и из-за отказа питания процесс записи может закончится неудачей в любое время, что означает, что некоторые из 512-байтовых секторов будут записаны, а некоторые нет. Чтобы защититься от таких сбоев, *перед* изменением фактической страницы на диске, Postgres Pro периодически записывает полные образы страниц на постоянное устройство хранения WAL. С помощью этого, во время восстановления после краха, Postgres Pro может восстановить из WAL страницы, которые записаны частично. Если у вас файловая система, которая защищена от частичной записи страниц (например, ZFS), вы можете выключить работу с образами страниц, выключив параметр full_page_writes. Батарея резервного питания (BBU) контроллера диска не защищает от частичной записи страниц, если не гарантируется, что данные записаны в BBU как полные (8kB) страницы.

Postgres Pro также защищает от некоторых видов повреждения данных на устройствах хранения, которые могут произойти из-за аппаратных ошибок или из-за дефектов поверхности с течением времени, например, при операциях чтения/записи во время сборки мусора.

- Каждая индивидуальная запись в WAL защищена с помощью контрольной суммы по алгоритму CRC-32 (32-bit), что позволяет судить о корректности данных в записи. Значение CRC устанавливается, когда мы пишем каждую запись WAL и проверяется в ходе восстановления после сбоя, восстановления из архива, и при репликации.

- Страницы данных в настоящее время не защищаются контрольными суммами по умолчанию, хотя полные образы страниц, записанные в WAL, будут защищены; смотрите initdb для деталей о включении в страницы данных информации о контрольных суммах.
- Для внутренних структур данных, таких как `pg_xact`, `pg_subtrans`, `pg_multixact`, `pg_serial`, `pg_notify`, `pg_stat`, `pg_snapshots` не ведётся расчёт контрольной суммы, равно как и для страниц, защищённых посредством полностраничной записи. Однако там, где такие структуры данных являются постоянными, записи WAL пишутся таким образом, чтобы после сбоя было возможно аккуратно повторить последние изменения, а эти записи WAL защищаются так же, как описано выше.
- Файлы каталога `pg_twophase` защищены с помощью контрольной суммы CRC-32.
- Временные файлы данных, используемые в больших SQL-запросах для сортировки, материализации и промежуточных результатов, в настоящее время не защищаются контрольной суммой, а изменения в этих файлах не отражаются в WAL.

Postgres Pro не защищает от корректируемых ошибок памяти; предполагается, что вы будете работать с памятью, которая использует промышленный стандарт коррекции ошибок (ECC, Error Correcting Codes) или лучшую защиту.

29.2. Журнал предзаписи (WAL)

Журнал предзаписи (WAL) — это стандартный метод обеспечения целостности данных. Детальное описание можно найти в большинстве книг (если не во всех) по обработке транзакций. Вкратце, основная идея WAL состоит в том, что изменения в файлах с данными (где находятся таблицы и индексы) должны записываться только после того, как эти изменения были занесены в журнал, т. е. после того как записи журнала, описывающие данные изменения, будут сохранены на постоянное устройство хранения. Если следовать этой процедуре, то записывать страницы данных на диск после подтверждения каждой транзакции нет необходимости, потому что мы знаем, что если случится сбой, то у нас будет возможность восстановить базу данных с помощью журнала: любые изменения, которые не были применены к страницам с данными, могут быть воссозданы из записей журнала. (Это называется восстановлением с воспроизведением, или REDO.)

> **Подсказка**
>
> Поскольку WAL восстанавливает содержимое файлов базы данных, журналируемая файловая система не является необходимой для надёжного хранения файлов с данными или файлов WAL. Фактически, журналирование может снизить производительность, особенно если журналирование заставляет сохранять *данные* файловой системы на диск. К счастью, такое сохранение при журналировании часто можно отключить с помощью параметров монтирования файловой системы, например, `data=writeback` для файловой системы ext3 в Linux. С другой стороны, с журналируемыми файловыми системами увеличивается скорость загрузки после сбоя.

Результатом использования WAL является значительное уменьшение количества запросов записи на диск, потому что для гарантии, что транзакция подтверждена, в записи на диск нуждается только файл журнала, а не каждый файл данных изменённый в результате транзакции. Файл журнала записывается последовательно и таким образом, затраты на синхронизацию журнала намного меньше, чем затраты на запись страниц с данными. Это особенно справедливо для серверов, которые обрабатывают много маленьких транзакций, изменяющих разные части хранилища данных. Таким образом, когда сервер обрабатывает множество мелких конкурентных транзакций, для подтверждения многих транзакций достаточно одного вызова `fsync` на файл журнала.

WAL также делает возможным поддержку онлайнового резервного копирования и восстановления на определённый момент времени, как описывается в Разделе 24.3. С помощью архивирования данных WAL поддерживается возврат к любому моменту времени, который доступен в данных

WAL: мы просто устанавливаем предыдущую физическую резервную копию базы данных и воспроизводим журнал WAL до нужного момента времени. Более того, физическая резервная копия не должна быть мгновенным снимком состояния баз данных — если она была сделана некоторое время назад, воспроизведение журнала WAL за этот период исправит все внутренние несоответствия.

29.3. Асинхронное подтверждение транзакций

Асинхронная фиксация — это возможность завершать транзакции быстрее, ценой того, что в случае краха СУБД последние транзакции могут быть потеряны. Для многих приложений такой компромисс приемлем.

Как описано в предыдущей части, подтверждение транзакции обычно *синхронное*: сервер ждёт сохранения записей WAL транзакции в постоянном хранилище, прежде чем сообщить клиенту об успешном завершении. Таким образом, клиенту гарантируется, что транзакция, которую подтвердил сервер, будет защищена, даже если сразу после этого произойдёт крах сервера. Однако для коротких транзакций данная задержка будет основной составляющей общего времени транзакции. В режиме асинхронного подтверждения сервер сообщает об успешном завершении сразу, как только транзакция будет завершена логически, прежде чем сгенерированные записи WAL фактически будут записаны на диск. Это может значительно увеличить производительность при выполнении небольших транзакций.

Асинхронное подтверждение транзакций приводит к риску потери данных. Существует короткое окно между отчётом о завершении транзакции для клиента и временем, когда транзакция реально подтверждена (т. е. гарантируется, что она не будет потеряна в случае краха сервера). Таким образом, асинхронное подтверждение транзакций не должно использоваться, если клиент будет выполнять внешние действия, опираясь на предположение, что транзакция будет сохранена. Например, банк конечно не должен использовать асинхронное подтверждение для транзакций в банкоматах, выдающих наличные. Но во многих случаях, таких как журналирование событий, столь серьёзная гарантия сохранности данных не нужна.

Риск потери данных при использовании асинхронного подтверждения транзакций — это не риск повреждения данных. Если случился крах СУБД, она будет восстановлена путём воспроизведения WAL до последней записи, которая была записана на диск. Таким образом, будет восстановлено целостное состояние СУБД, но любые транзакции, которые ещё не были сохранены на диск, в этом состоянии не будут отражены. Чистый эффект будет заключаться в потере нескольких последних транзакций. Поскольку транзакции воспроизводятся в том же порядке, в котором подтверждались, воспроизведение не нарушает целостность — например, если транзакция «B» выполнила изменения, которые влияют на предыдущую транзакцию «A», то не может быть такого, что изменения, выполненные «A» были потеряны, а изменения, внесённые «B» сохранены.

Пользователь может выбрать режим подтверждения для каждой транзакции, так что возможен конкурентный запуск транзакций в синхронном и асинхронном режиме. Это позволяет достичь гибкого компромисса между производительностью и конечно надёжностью транзакций. Режим подтверждения транзакций управляется параметром synchronous_commit, который может быть изменён любым из способов, пригодным для установки параметров конфигурации. Режим, используемый для какой-либо конкретной транзакции, зависит от значения `synchronous_commit`, которое действует на момент начала этой транзакции.

Некоторые команды, например `DROP TABLE`, принудительно запускают синхронное подтверждение транзакции, независимо от значения `synchronous_commit`. Это сделано для того, чтобы иметь уверенность в целостности данных между файловой системой сервера и логическим состоянием базы данных. Команды, которые поддерживают двухфазное подтверждение транзакций, такие как `PREPARE TRANSACTION`, также всегда синхронные.

Если во время окна риска между асинхронным подтверждением транзакции и сохранением на диск записей WAL, происходит крах СУБД, то изменения, сделанные во время этой транзакции *будут* потеряны. Продолжительность окна риска ограничена, потому что фоновый процесс («WAL writer»), сохраняет не записанные записи WAL на диск каждые wal_writer_delay миллисекунд.

Фактически, максимальная продолжительность окна риска составляет трёхкратное значение `wal_writer_delay`, потому что WAL writer разработан так, чтобы сразу сохранять целые страницы во время периодов занятости.

> **Внимание**
>
> Режим немедленного завершения работы (immediate) эквивалентен краху сервера и приведёт, таким образом, к потере всех не сохранённых асинхронных транзакций.

Асинхронное подтверждение транзакций предоставляет поведение, которое отличается от того, что соответствует установке параметра fsync = off. Настройка `fsync` касается всего сервера и может изменить поведение всех транзакций. Она выключает всю логику внутри Postgres Pro, которая пытается синхронизировать запись отдельных порций в базу данных и, таким образом, крах системы (обусловленный отказом аппаратного обеспечения или операционной системы, который не является сбоем самой СУБД Postgres Pro) может в результате привести к повреждению состояния базы данных. Во многих случаях, асинхронное подтверждение транзакций предоставляет лучшую производительность, чем то, что можно получить выключением `fsync`, но без риска повреждения данных.

commit_delay также выглядит очень похоже на асинхронное подтверждение транзакций, но по сути это является методом синхронного подтверждения транзакций (фактически, во время асинхронных транзакций `commit_delay` игнорируется). `commit_delay` приводит к задержке только перед тем, как синхронная транзакция пытается записать данные WAL на диск, в надежде, что одиночная запись, выполняемая на одну такую транзакцию, сможет также обслужить другие транзакции, которые подтверждаются приблизительно в это же время. Установку этого параметра можно рассматривать как способ увеличения промежутка времени, в течение которого транзакции группируются для единовременной записи на диск. Это распределяет стоимость записи между несколькими транзакциями.

29.4. Настройка WAL

Существует несколько конфигурационных параметров относящихся к WAL, которые влияют на производительность СУБД. Далее рассказывается об их использовании. Общую информацию об установке параметров конфигурации сервера смотрите в Главе 18.

Контрольные точки— это точки в последовательности транзакций, в которых гарантируется, что файлы с данными и индексами были обновлены всей информацией записанной перед контрольной точкой. Во время контрольной точки, все «грязные» страницы данных, находящиеся в памяти, сохраняются на диск, а в файл журнала записывается специальная запись контрольной точки. (Сделанные изменения были перед этим записаны в файлы WAL.) В случае краха процедура восстановления ищет последнюю запись контрольной точки, чтобы определить эту точку в журнале (называемую записью REDO), от которой процедура должна начать операцию воспроизведения изменений. Любые изменения файлов данных перед этой точкой гарантированно находятся уже на диске. Таким образом, после контрольной точки, сегменты журнала, которые предшествуют записи воспроизведения, больше не нужны и могут быть удалены или пущены в циклическую перезапись. (Когда архивирование WAL будет завершено, сегменты журнала должны быть архивированы перед их удалением или циклической перезаписи.)

Запись всех «грязных» страниц данных из памяти на диск, которая требуется для контрольной точки, может вызвать значительную нагрузку на дисковый ввод/вывод. По этой причине, активность записи по контрольной точке регулируется так, что ввод/вывод начинается при старте контрольной точки и завершается перед стартом следующей контрольной точки; это минимизирует потерю производительности во время прохождения контрольных точек.

Отдельный серверный процесс контрольных точек автоматически выполняет контрольные точки с заданной частотой. Контрольные точки производятся каждые checkpoint_timeout секунд либо при приближении к пределу max_wal_size, если это имеет место раньше. Значения по умолчанию:

5 минут и 1 Гбайт, соответственно. Если после предыдущей контрольной точки новые записи WAL не добавились, следующие контрольные точки будут пропущены, даже если проходит время checkpoint_timeout. (Если вы применяете архивацию WAL и хотите установить нижний предел для частоты архивации, чтобы ограничить потенциальную потерю данных, вам следует настраивать параметр archive_timeout, а не параметры контрольных точек.) Также можно выполнить контрольную точку принудительно, воспользовавшись SQL-командой CHECKPOINT.

Уменьшение значений checkpoint_timeout и/или max_wal_size приводит к учащению контрольных точек. Это позволяет ускорить восстановление после краха (поскольку для воспроизведения нужно меньше данных), но с другой стороны нужно учитывать дополнительную нагрузку, возникающую вследствие более частого сброса «грязных» страниц данных на диск. Если включён режим full_page_writes (по умолчанию это так), нужно учесть и ещё один фактор. Для обеспечения целостности страницы данных, при первом изменении страницы данных после контрольной точки эта страница записывается в журнал целиком. В данном случае, чем меньше интервал между контрольными точками, тем больше объём записи в журнал WAL, так что это частично дискредитирует идею уменьшения интервала записи, и в любом случае приводит к увеличению объёма обмена с диском

Контрольные точки довольно дороги с точки зрения ресурсов, во-первых, потому что они требуют записи всех «грязных» буферов из памяти на диск, и во-вторых потому что они создают дополнительный трафик WAL, о чём говорилось выше. Таким образом, будет благоразумным установить параметры контрольных точек так, чтобы контрольные точки не выполнялись слишком часто. Для простой проверки параметров контрольной точки можно установить параметр checkpoint_warning. Если промежуток времени между контрольными точками будет меньше чем количество секунд, заданное параметром checkpoint_warning, то в журнал сервера будет выдано сообщение с рекомендацией увеличить max_wal_size. Эпизодическое появление такого сообщения не является поводом для беспокойства. Но если оно появляется часто, необходимо увеличить значения параметров управления контрольными точками. Массовые операции, такие как COPY с большим объёмом данных, могут привести к появлению нескольких таких предупреждений, если вы не установили max_wal_size достаточно большим.

Чтобы избежать «заваливания» системы ввода/вывода при резкой интенсивной записи страниц, запись «грязных» буферов во время контрольной точки растягивается на определённый период времени. Этот период управляется параметром checkpoint_completion_target, который задаётся как часть интервала контрольной точки. Скорость ввода/вывода подстраивается так, чтобы контрольная точка завершилась к моменту истечения заданной части от checkpoint_timeout секунд или до превышения max_wal_size, если оно имеет место раньше. Со значением 0.5, заданным по умолчанию, можно ожидать, что Postgres Pro завершит процедуру контрольной точки примерно за половину времени до начала следующей. В системе, которая работает практически на пределе мощности ввода/вывода в обычном режиме, есть смысл увеличить checkpoint_completion_target, чтобы уменьшить нагрузку ввода/вывода, связанную с контрольными точками. Но с другой стороны, растягивание контрольных точек влияет на время восстановления, так как для восстановления нужно будет задействовать большее количество сегментов WAL. Хотя в checkpoint_completion_target можно задать значение вплоть до 1.0, лучше выбрать значение меньше (по крайней мере, не больше 0.9), так как при контрольных точках выполняются и некоторые другие операции, помимо записи «грязных» буферов. Со значением 1.0 контрольные точки, скорее всего, не будут завершаться вовремя, что приведёт к потере производительности из-за неожиданных колебаний требуемого количества сегментов WAL.

На платформах Linux и POSIX параметр checkpoint_flush_after позволяет принудить ОС к сбросу страниц, записываемых во время контрольной точки, при накоплении заданного количества байт. Если его не настроить, эти страницы могут оставаться в кеше страниц ОС, что повлечёт затормаживание при выполнении fsync в конце контрольной точки. Этот параметр часто помогает уменьшить задержки транзакций, но может оказать и негативное влияние на производительность; особенно, когда объём нагрузки больше shared_buffers, но меньше кеша страниц в ОС.

Число файлов сегментов WAL в каталоге pg_wal зависит от min_wal_size, max_wal_size и объёма WAL, сгенерированного в предыдущих циклах контрольных точек. Когда старые файлы сегментов оказываются не нужны, они удаляются или перерабатываются (то есть переименовываются,

чтобы стать будущими сегментами в нумерованной последовательности). Если вследствие кратковременного скачка интенсивности записи в журнал, предел max_wal_size превышается, ненужные файлы сегментов будут удаляться, пока система не опустится ниже этого предела. Оставаясь ниже этого предела, система перерабатывает столько файлов WAL, сколько необходимо для покрытия ожидаемой потребности до следующей контрольной точки, и удаляет остальные. Эта оценка базируется на скользящем среднем числа файлов WAL, задействованных в предыдущих циклах контрольных точек. Скользящее среднее увеличивается немедленно, если фактическое использование превышает оценку, так что в нём в некоторой степени накапливается пиковое использование, а не среднее. Значение min_wal_size ограничивает снизу число файлов WAL, которые будут переработаны для будущего использования; такой объём WAL всегда будет перерабатываться, даже если система простаивает и оценка использования говорит, что нужен совсем небольшой WAL.

Вне зависимости от max_wal_size, в количестве wal_keep_segments + 1 самые последние файлы WAL сохраняются в любом случае. Кроме того, если применяется архивация WAL, старые сегменты не могут быть удалены или переработаны, пока они не будут заархивированы. Если WAL архивируется медленнее, чем генерируется, либо если команда archive_command постоянно даёт сбои, старые файлы WAL будут накапливаться в pg_wal, пока ситуация не будет разрешена. Медленно работающий или отказавший ведомый сервер, использующий слот репликации, даст тот же эффект (см. Подраздел 25.2.6).

В режиме восстановления архива или горячего резерва сервер периодически выполняет *точки перезапуска*, которые похожи на контрольные точки в обычном режиме работы: сервер принудительно сбрасывает своё состояние на диск, обновляет файл pg_control, чтобы показать, что уже обработанные данные WAL не нужно сканировать снова, и затем перерабатывает все старые файлы сегментов журнала в каталоге pg_wal. Точки перезапуска не могут выполняться чаще, чем контрольные точки на главном сервере, так как они могут происходить только в записях контрольных точек. Точка перезапуска производится, когда достигается запись контрольной точки и после предыдущей точки перезапуска прошло не меньше checkpoint_timeout секунд или размер WAL может превысить max_wal_size. Однако из-за того, что на время выполнения точек перезапуска накладываются ограничения, max_wal_size часто превышается при восстановлении, вплоть до объёма WAL, записываемого в цикле между контрольными точками. (Значение max_wal_size никогда и не было жёстким пределом, так что всегда следует оставлять приличный запас сверху, чтобы не остаться без свободного места на диске.)

Наиболее часто используются две связанные с WAL внутренние функции: XLogInsertRecord и XLogFlush. XLogInsertRecord применяется для добавления записи в буферы WAL в разделяемой памяти. Если места для новой записи недостаточно, XLogInsertRecord придётся записать (переместить в кеш ядра) несколько заполненных буферов WAL. Это нежелательно, так как XLogInsertRecord используется при каждом изменении в базе данных на низком уровне (например, при добавлении строки) в момент, когда установлена исключительная блокировка задействованных страниц данных, поэтому данная операция должна быть максимально быстрой. Что ещё хуже, запись буферов WAL может также повлечь создание нового сегмента журнала, что займёт ещё больше времени. Обычно буферы WAL должны записываться и сохраняться на диске в функции XLogFlush, которая вызывается, по большей части, при фиксировании транзакции, чтобы результаты транзакции сохранились в надёжном хранилище. В системах с интенсивной записью в журнал вызовы XLogFlush могут иметь место не так часто, чтобы XLogInsertRecord не приходилось производить запись. В таких системах следует увеличить число буферов WAL, изменив параметр wal_buffers. Когда включён режим full_page_writes и система очень сильно загружена, увеличение wal_buffers поможет сгладить скачки во времени ответа в период сразу после каждой контрольной точки.

Параметр commit_delay определяет, на сколько микросекунд будет засыпать ведущий процесс группы, записывающий в журнал, после получения блокировки в XLogFlush, пока подчинённые формируют очередь на запись. Во время этой задержки другие серверные процессы смогут добавлять записи в WAL буферы журнала, чтобы все эти записи сохранились на диск в результате одной операции синхронизации, которую выполнит ведущий. Ведущий процесс не засыпает, если отключён режим fsync, либо число сеансов с активными транзакциями меньше commit_siblings,

так как маловероятно, что какой-либо другой сеанс зафиксирует транзакцию в ближайшее время. Заметьте, что на некоторых платформах, разрешение этого таймера сна составляет 10 миллисекунд, так что любое значение параметра commit_delay от 1 до 10000 микросекунд будет действовать одинаково. Кроме того, в некоторых системах состояние сна может продлиться несколько дольше, чем требует параметр.

Так как цель commit_delay состоит в том, чтобы позволить стоимости каждой операции синхронизации амортизироваться через параллельную фиксацию транзакций (потенциально за счёт задержки транзакции), необходимо определить количество той стоимости, прежде чем урегулирование сможет быть выбрано разумно. Чем выше стоимость, тем более эффективный будет commit_delay в увеличении пропускной способности транзакций в какой-то степени. Программа pg_test_fsync может использоваться, чтобы измерить среднее время в микросекундах, которое занимает одиночная работа сброса WAL на диск. Значение половины среднего времени сообщаемого программой рекомендуется в качестве отправной точки для использования значения в параметре commit_delay при оптимизации для конкретного объёма работы, и говорит о том, сколько нужно времени для синхронизации сброса единственной операции записи 8 Кбайт. Настройка параметра commit_delay особенно полезна в случае хранения WAL в хранилище с высокоскоростными дисками, такими как твердотельные накопители (SSD) или RAID-массивы с кешем записи и аварийным питанием на батарее; но это определённо должно тестироваться на репрезентативной рабочей нагрузке. Более высокие значения commit_siblings должны использоваться в таких случаях, тогда как меньшие значения commit_siblings часто полезны на носителях с большими задержками. Обратите внимание на то, что увеличение значения параметра commit_delay может увеличить задержку транзакции настолько, что пострадает общая производительность транзакций.

Даже если commit_delay равен нулю (значение по умолчанию), групповая фиксация все равно может произойти, но группа будет состоять только из тех сеансов, которым понадобилось сбросить записи о фиксации на диск за то время, пока происходил предыдущий сброс. Чем больше сеансов, тем чаще это происходит даже при нулевом commit_delay, поэтому увеличение этого параметра может и не оказать заметного действия. Установка commit_delay имеет смысл в двух случаях: (1) когда несколько транзакций одновременно фиксируют изменения, (2) либо когда частота фиксаций ограничена пропускной способностью дисковой подсистемы. Однако при задержке из-за низкой скорости вращения диска, эта настройка может оказаться полезной даже всего при двух сеансах.

Параметр wal_sync_method определяет, как Postgres Pro будет обращаться к ядру, чтобы принудительно сохранить WAL на диск. Все методы должны быть одинаковыми в плане надёжности, за исключением fsync_writethrough, который может иногда принудительно сбрасывать кеш диска, даже если другие методы не делают этого. Однако какой из них самый быстрый, во многом определяется платформой; вы можете протестировать скорость, используя модуль pg_test_fsync. Обратите внимание, что данный параметр не имеет значения, если fsync выключен.

Включение параметра конфигурации wal_debug (предоставляется, если Postgres Pro был скомпилирован с его поддержкой) будет приводить к тому, что все вызовы связанных с WAL функций XLogInsertRecord и XLogFlush будут протоколироваться в журнале сервера. В будущем данный параметр может быть заменён более общим механизмом.

29.5. Внутреннее устройство WAL

WAL включается автоматически; от администратора не требуется никаких действий за исключением того, чтобы убедиться, что выполнены требования WAL к месту на диске, и что выполнены все необходимые действия по тонкой настройке (см. Раздел 29.4).

Записи WAL добавляются в журналы WAL по мере поступления. Позицию добавления в журнал определяет значение LSN (Log Sequence Number, Последовательный номер в журнале), представляющее собой смещение в байтах внутри журнала, монотонно увеличивающееся с каждой новой записью. Значения LSN возвращаются с типом данных pg_lsn. Сравнивая эти значения,

можно вычислить объём данных WAL между ними, так что они могут быть полезны для вычисления прогресса при репликации и восстановлении.

Журналы WAL хранятся в виде набора файлов сегментов в каталоге pg_wal, находящемся в каталоге данных. Эти файлы обычно имеют размер 16 Мбайт каждый. Каждый файл сегмента разделяется на страницы, обычно по 8 Кбайт. Содержимое записи журнала зависит от типа события, которое сохраняется в журнале. Файлы сегментов имеют имена-номера, которые начинаются с `000000010000000000000001` и последовательно увеличиваются. Зацикливание этих номеров не предусмотрено, но для использования всех доступных номеров потребуется очень много времени.

Имеет смысл размещать журналы WAL на другом диске, отличном от того, где находятся основные файлы базы данных. Для этого можно переместить каталог `pg_wal` в другое место (разумеется, когда сервер остановлен) и создать символическую ссылку из исходного места на перемещённый каталог.

Для WAL важно, чтобы запись в журнал выполнялась до изменений данных в базе. Но этот порядок могут нарушить дисковые устройства, которые ложно сообщают ядру об успешном завершении записи, хотя фактически они только выполнили кеширование данных и пока не сохранили их на диск. Сбой питания в такой ситуации может привести к неисправимому повреждению данных. Администраторы должны убедиться, что диски, где хранятся файлы журналов WAL Postgres Pro, не выдают таких ложных сообщений ядру. (См. Раздел 29.1.)

После выполнения контрольной точки и сброса журнала позиция контрольной точки сохраняется в файл `pg_control`. Таким образом, при старте восстановления сервер сперва читает файл `pg_control` и затем запись контрольной точки; затем он выполняет операцию REDO, сканируя вперёд от позиции в журнале, обозначенной в записи контрольной точки. Поскольку полное содержимое страниц данных сохраняется в журнале в первой странице после контрольной точки (предполагается, что включён режим full_page_writes), все страницы, изменённые с момента контрольной точки, будут восстановлены в целостном состоянии.

В случае, если файл `pg_control` повреждён, мы должны поддерживать возможность сканирования существующих сегментов журнала в обратном порядке — от новых к старым — чтобы найти последнюю контрольную точку. Это пока не реализовано. `pg_control` является достаточно маленьким файлом (меньше, чем одна дисковая страница), который не должен попадать под проблему частичной записи и на момент написания данной документации, не было ни одного сообщения о сбоях СУБД исключительно из-за невозможности чтения самого файла `pg_control`. Таким образом, хотя теоретически это является слабым местом, на практике проблем с `pg_control` не обнаружено.

Глава 30. Логическая репликация

Логическая репликация — это метод репликации объектов данных и изменений в них, использующий репликационные идентификаторы (обычно это первичный ключ). Мы называем такую репликацию «логической», в отличие от физической, которая построена на точных адресах блоков и побайтовом копировании. Postgres Pro поддерживает оба механизма одновременно; см. Главу 25. Логическая репликация позволяет более детально управлять репликацией данных и аспектами безопасности.

В логической репликации используется модель *публикаций/подписок* с одним или несколькими *подписчиками*, которые подписываются на одну или несколько *публикаций* на *публикующем* узле. Подписчики получают данные из публикаций, на которые они подписаны, и могут затем повторно опубликовать данные для организации каскадной репликации или более сложных конфигураций.

Логическая репликация таблицы обычно начинается с создания снимка данных в публикуемой базе данных и копирования её подписчику. После этого изменения на стороне публикации передаются подписчику в реальном времени, когда они происходят. Подписчик применяет изменения в том же порядке, что и узел публикации, так что для публикаций в рамках одной подписки гарантируется транзакционная целостность. Этот метод репликации данных иногда называется транзакционной репликацией.

Типичные сценарии использования логической репликации:

- Передача подписчикам инкрементальных изменений в одной базе данных или подмножестве базы данных, когда они происходят.
- Срабатывание триггеров для отдельных изменений, когда их получает подписчик.
- Объединение нескольких баз данных в одну (например, для целей анализа).
- Репликация между разными основными версиями Postgres Pro.
- Репликация между экземплярами Postgres Pro на разных платформах (например, с Linux на Windows)
- Предоставление доступа к реплицированным данным другим группам пользователей.
- Разделение подмножества базы данных между несколькими базами данных.

База данных подписчика функционирует так же, как и любой другой экземпляр базы Postgres Pro, и может стать публикующей, если создать публикации в ней. Когда подписчик действует как исключительно читающее приложение, никаких конфликтов с одной подпиской не будет. Но они могут возникнуть, если в тот же набор таблиц производят запись какие-либо приложения или другие подписчики.

30.1. Публикация

Публикация может быть определена на любом ведущем сервере физической репликации. Сервер, на котором определяется публикация, называется *публикующим*. Публикация — это набор изменений, выделяемых в таблице или в группе таблиц (он также может называться набором репликации). Публикация существует только в одной базе данных.

Публикации отличаются от схем и они никак не влияют на доступ к таблице. Если требуется, каждую таблицу можно включить в несколько публикаций. В настоящее время публикации могут содержать только таблицы. Объекты в них нужно добавлять явным образом, если только публикация не создана для всех таблиц (с указанием `ALL TABLES`).

Публикации могут ограничивать набор публикуемых изменений, выбирая любое сочетание операций из `INSERT`, `UPDATE` и `DELETE`, подобно тому как триггеры могут срабатывать для разных типов событий. По умолчанию реплицируются все типы операций.

Чтобы можно было реплицировать операции `UPDATE` и `DELETE`, в публикуемой таблице должен быть настроен «репликационный идентификатор» для нахождения соответствующих строк для

изменения или удаления на стороне подписчика. По умолчанию это первичный ключ, если он создан. Также репликационным идентификатором можно назначить другой уникальный индекс (с некоторыми дополнительными условиями). Если в таблице нет подходящего ключа, в качестве репликационного идентификатора можно задать «full», что будет означать, что ключом будет вся строка. Это, однако, очень неэффективно и должно применяться как крайняя мера, если другого решения нет. Если на стороне публикации выбран репликационный идентификатор, отличный от «full», то идентификатор, состоящий из того же или меньшего количества столбцов, также должен быть определён на стороне подписчика. Подробнее о назначении репликационного идентификатора рассказывается в `REPLICA IDENTITY`. Если в публикацию, в которой реплицируются операции `UPDATE` и `DELETE`, добавляется таблица без репликационного идентификатора, то последующие команды `UPDATE` и `DELETE` на стороне публикации вызовут ошибку. Команды `INSERT` могут выполняться вне зависимости от такого идентификатора.

У каждой публикации может быть множество подписчиков.

Публикация создаётся командой CREATE PUBLICATION и может быть впоследствии изменена или удалена с помощью соответствующих команд.

В публикации можно динамически добавлять или удалять отдельные таблицы, используя команду ALTER PUBLICATION. Операции `ADD TABLE` и `DROP TABLE` являются транзакционными, так что репликация таблицы будет начата или закончена с определённым снимком только после фиксации транзакции.

30.2. Подписка

Подписка — это принимающая сторона логической репликации. Узел, на котором определяется подписка, называется *подписчиком*. В свойствах подписки определяется подключение к другой базе данных и набор публикаций (из одной или нескольких), на которые подписчик хочет подписаться.

База данных подписчика работает так же, как и экземпляр любой другой базы Postgres Pro, и может быть публикующей для других баз, если в ней определены собственные подписки.

Узел подписчика может подписываться на несколько подписок, если требуется. В одной паре публикующий сервер/подписчик могут быть определены несколько подписок, но при этом нужно позаботиться о том, чтобы публикуемые объекты в разных подписках не перекрывались.

Изменения в каждой подписке будут приходить через один слот репликации (см. Подраздел 25.2.6). Дополнительные слоты репликации могут потребоваться для начальной синхронизации уже существующих данных таблиц.

Подписка логической репликации может представлять собой ведомый узел для синхронной репликации (см. Подраздел 25.2.8). В этом случае именем ведомого узла по умолчанию будет имя подписки. Другое имя можно задать в свойстве `application_name` в строке подключения для данной подписки.

Подписки могут быть выгружены командой `pg_dump`, если её выполняет суперпользователь. В противном случае выдаётся предупреждение и подписки пропускаются, так как обычным пользователям не разрешено читать всю информацию о подписках из каталога `pg_subscription`.

Подписки добавляются командой CREATE SUBSCRIPTION и могут быть остановлены/возобновлены в любой момент командой ALTER SUBSCRIPTION, а также удалены командой DROP SUBSCRIPTION.

Когда подписка удаляется и пересоздаётся, информация о синхронизации теряется. Это означает, что после этого данные необходимо синхронизировать заново.

Определения схемы не реплицируются, а публикуемые таблицы должны существовать в базе подписчика. Объектами репликации могут быть только обычные таблицы. Так, например, нельзя произвести репликацию в представление.

Таблицы публикации сопоставляются с таблицами подписчика по полностью заданным именам таблиц. Репликация в таблицы с другими именами на стороне подписчика не поддерживается.

Столбцы таблиц также сопоставляются по именам. Порядок столбцов в таблице подписчика может отличаться от порядка столбцов в публикации. Также могут не совпадать типы столбцов; достаточно только возможности преобразования текстового представления данных в целевой тип. Например, данные столбца типа `integer` могут реплицироваться в столбец типа `bigint`. Целевая таблица может также содержать дополнительные столбцы, отсутствующие в публикуемой таблице. Такие столбцы будут заполнены значениями по умолчанию, заданными в определении целевой таблицы.

30.2.1. Управление слотами репликации

Как было сказано ранее, каждая (активная) подписка получает изменения из слота репликации на удалённой стороне (стороне публикации). Обычно удалённый слот репликации создаётся автоматически, когда подписка создаётся командой `CREATE SUBSCRIPTION`, и удаляется автоматически, когда она удаляется командой `DROP SUBSCRIPTION`. Однако в некоторых ситуациях может быть полезно или необходимо манипулировать подпиской и нижележащим слотом по отдельности. Например, возможны такие сценарии:

- При создании подписки слот репликации может уже существовать. В этом случае подписку можно создать с параметром `create_slot = false`, чтобы она была связана с существующим слотом.
- При создании подписки удалённый узел может быть недоступен или находиться в нерабочем состоянии. В этом случае подписку можно создать с указанием `connect = false`. При этом подключение к удалённому узлу не будет устанавливаться. Этот вариант использует pg_dump. Чтобы активировать такую подписку впоследствии, удалённый слот репликации нужно будет создать вручную.
- При ликвидации публикации может потребоваться сохранить слот репликации. Например, это полезно, когда нужно перенести базу данных подписчика на другой узел и активировать её там. В этом случае разорвите связь подписки со слотом, используя команду `ALTER SUBSCRIPTION`, прежде чем удалять подписку.
- При ликвидации подписки удалённый узел может быть недоступен. В этом случае разорвите связь подписки со слотом, используя команду `ALTER SUBSCRIPTION`, прежде чем пытаться удалять подписку. Если удалённый экземпляр базы данных прекратил существование, больше никакие действия не требуются. Если же экземпляр удалённой базы данных просто оказался недоступным, слот репликации нужно будет удалить вручную; в противном случае публикующий сервер продолжит сохранять WAL и может в конце концов заполнить всё место на диске. Такие случаи заслуживают самого серьёзного разбирательства.

30.3. Конфликты

Логическая репликация работает подобно обычным операциям DML в том смысле, что данные будут изменены, даже если они независимо изменялись на стороне подписчика. Если входящие данные нарушат какие-либо ограничения, репликация остановится. Эта ситуация называется *конфликтом*. При репликации операций `UPDATE` или `DELETE` отсутствие данных не вызывает конфликта, так что такие операции просто пропускаются.

В случае конфликта выдаётся ошибка и репликация останавливается; разрешить возникшую проблему пользователь должен вручную. Подробности конфликта можно найти в журнале сервера-подписчика.

Разрешение может заключаться либо в изменении данных на стороне подписчика, чтобы они не конфликтовали с приходящим изменением, либо в пропуске транзакции, конфликтующей с существующими данными. Пропустить транзакцию можно, вызвав функцию `pg_replication_origin_advance()`, которой передаётся в *node_name* соответствующее имя подписки, а также позиция. Текущие позиции источников можно увидеть в системном представлении `pg_replication_origin_status`.

30.4. Ограничения

Логическая репликация в настоящее время имеет ограничения и недостатки, описанные ниже. Они могут быть устранены в будущих выпусках.

- Схема базы данных и команды DDL не реплицируются. Изначальную схему можно скопировать, воспользовавшись командой `pg_dump --schema-only`. Последующие изменения схемы необходимо будет синхронизировать вручную. (Заметьте, однако, что схемы не обязательно должны быть абсолютно идентичными на обеих сторонах репликации.) Если определения схемы в исходной базе данных меняются, логическая репликация работает надёжно — когда данные после изменения схемы прибывают на сторону подписчика, но не вписываются в схему его таблиц, выдаётся ошибка, требующая обновления схемы. Во многих случаях возникновение таких ошибок можно предупредить, сначала применяя дополняющие изменения на подписчике.
- Данные последовательностей не реплицируются. Данные в столбцах serial или столбцах идентификации, выдаваемые последовательностями, конечно, будут реплицированы в составе таблицы, но сама последовательность на подписчике будет сохранять стартовое значение. Если подписчик используется в качестве базы только для чтения, обычно это не является проблемой. Если же, однако, предусматривается возможность переключения на базу подписчика некоторым образом, текущие значения в этих последовательностях нужно будет обновить, либо скопировав текущие данные из базы публикации (вероятно, с применением `pg_dump`), либо выбрав достаточно большие значения из самих таблиц.
- Команды `TRUNCATE` не реплицируются. Разумеется, это можно обойти, выполняя вместо них команды `DELETE`. Чтобы предотвратить случайное выполнение `TRUNCATE`, вы можете отозвать право `TRUNCATE` для реплицируемых таблиц.
- Большие объекты (см. Главу 32) не реплицируются. Это ограничение нельзя обойти никак, кроме как хранить данные в обычных таблицах.
- Реплицировать данные возможно только из базовых таблиц в базовые таблицы. То есть таблицы на стороне публикации и на стороне подписки должны быть обычными, а не представлениями, мат. представлениями, секционированными или сторонними таблицами. Это означает, что вы можете реплицировать секции одну в одну, но реплицировать данные в таблицы, секционированные по-другому, нельзя. При попытке реплицировать таблицы, отличные от базовых, будет выдана ошибка.

30.5. Архитектура

Логическая репликация начинается с копирования снимка данных в базе данных публикации. По завершении этой операции изменения на стороне публикации передаются подписчику в реальном времени, когда они происходят. Подписчик применяет изменения в том же порядке, в каком они вносятся на узле публикации, так что для публикаций в рамках одной подписки гарантируется транзакционная целостность.

Логическая репликация построена по схеме, подобной физической потоковой репликации (см. Подраздел 25.2.5). Она реализуется процессами «walsender» (передачи WAL) и «apply» (применения). Процесс walsender запускает логическое декодирование (описанное в Главе 46) WAL и загружает стандартный модуль логического декодирования (pgoutput). Этот модуль преобразует изменения, считываемые из WAL, в протокол логической репликации (см. Раздел 50.5) и отфильтровывает данные согласно спецификации публикации. Затем данные последовательно передаются по протоколу логической репликации рабочему процессу применения изменений, который сопоставляет данные с логическими таблицами и применяет отдельные изменения по мере их поступления, сохраняя транзакционный порядок.

Процесс применения изменений в базе данных подписчика всегда выполняется со значением `session_replication_role`, равным `replica`, что влечёт соответствующие эффекты для триггеров и ограничений.

Процесс применения логической репликации в настоящее время вызывает только триггеры уровня строк, но не триггеры операторов. Однако начальная синхронизация таблицы реализована как команда `COPY` и поэтому вызывает триггеры для `INSERT` и уровня строк, и уровня оператора.

30.5.1. Начальный снимок

Начальные данные существующих таблиц в подписке помещаются в снимок и копируются в параллельном экземпляре процесса применения особого вида. Этот процесс создаёт собственный слот репликации и производит копирование существующих данных. Когда существующие данные будут скопированы, этот рабочий процесс переходит в режим синхронизации, в котором таблица приводится в синхронизированное состояние для основного процесса применения, то есть передаёт все изменения, произошедшие во время начального копирования данных, используя стандартную логическую репликацию. По завершении синхронизации управление репликацией этой таблицы возвращается главному процессу, который продолжает репликацию в обычном режиме.

30.6. Мониторинг

Так как логическая репликация построена по схеме, подобной физической потоковой репликации, мониторинг публикующего узла подобен мониторингу ведущего сервера при физической репликации (см. Подраздел 25.2.5.2).

Информацию о подписке для мониторинга можно получить в представлении `pg_stat_subscription`. Это представление содержит по одной строке для каждого рабочего процесса подписчика. В зависимости от состояния подписки, с ней может быть связано ноль или более активных рабочих процессов.

Обычно для включённой подписки работает только один процесс применения. Для отключённой или нарушенной подписки это представление будет содержать ноль строк. Если выполняется начальная синхронизация данных для каких-либо таблиц, для этих таблиц будут показаны дополнительные рабочие процессы, производящие синхронизацию.

30.7. Безопасность

Пользователь, имеющий право изменения схемы таблиц на стороне подписчика, может выполнить произвольный код как суперпользователь. Ограничьте круг владельцев и ролей, имеющих право `TRIGGER` для таких таблиц, доверенными ролями. Более того, в базе, где недоверенные пользователи могут создавать таблицы, включайте в публикацию только таблицы по списку. Другими словами, создавайте подписку `FOR ALL TABLES`, только когда суперпользователи доверяют всем пользователям, имеющим право создавать не временные таблицы на стороне публикации или подписки.

Роль, используемая для подключения репликации, должна иметь атрибут `REPLICATION` (или быть суперпользователем). Если у роли отсутствуют свойства `SUPERUSER` и `BYPASSRLS`, при репликации могут выполняться политики защиты строк, определённые на стороне публикации. Если эта роль не может доверять владельцам всех таблиц, добавьте в строку подключения `options=-crow_security=off`; если владелец таблицы добавит политику защиты строк позже, при таком значении параметра репликация остановится, но политика выполняться не будет. Доступ для этой роли должен быть настроен в `pg_hba.conf`, и эта роль также должна иметь атрибут `LOGIN`.

Чтобы иметь возможность скопировать исходные данные таблицы, роль, используемая для соединения репликации, должна иметь право `SELECT` в публикуемой таблице (или быть суперпользователем).

Чтобы создать публикацию, пользователь должен иметь право `CREATE` в базе данных.

Чтобы добавлять таблицы в публикацию, пользователь должен иметь права владельца для этих таблиц. Создавать публикации, публикующие все таблицы автоматически, разрешено только суперпользователям.

Создавать подписки разрешено только суперпользователям.

Процесс применения изменений подписки будет выполняться в локальной базе с правами суперпользователя.

Права проверяются только один раз при установлении подключения для репликации. Они не перепроверяются при чтении каждой записи изменения с публикующего сервера и не перепроверяются при применении каждого изменения.

30.8. Параметры конфигурации

Для осуществления логической репликации необходимо установить несколько параметров конфигурации.

На публикующем сервере параметр `wal_level` должен иметь значение `logical`, а в `max_replication_slots` должно быть задано число не меньше ожидаемого числа подписчиков плюс некоторый резерв для синхронизации таблиц. А в `max_wal_senders` должно быть значение как минимум равное `max_replication_slots` плюс число возможных физических реплик, работающих одновременно.

Также на стороне подписчика необходимо установить параметр `max_replication_slots`, задающий число источников репликации, которые могут отслеживаться. В данном случае он должен быть не меньше числа подписок, на которые будет подписываться данный подписчик. В `max_logical_replication_workers` необходимо установить значение не меньше числа подписок плюс некоторый резерв для синхронизации таблиц. Кроме того, может потребоваться изменить `max_worker_processes`, чтобы это число включало дополнительные рабочие процессы для репликации (как минимум `max_logical_replication_workers + 1`). Заметьте, что некоторые расширения и параллельные запросы также занимают слоты из числа `max_worker_processes`.

30.9. Быстрая настройка

Сначала установите параметры конфигурации в `postgresql.conf`:

```
wal_level = logical
```

Другие необходимые параметры по умолчанию имеют значения, подходящие для базовой настройки.

В файл `pg_hba.conf` необходимо внести изменения, чтобы разрешить репликацию (конкретные значения будут зависеть от фактической конфигурации сети и имени пользователя, с которым вы будете подключаться):

```
host     all     repuser     0.0.0.0/0     md5
```

Затем в базе данных публикации выполните:

```
CREATE PUBLICATION mypub FOR TABLE users, departments;
```

И в базе данных подписчика:

```
CREATE SUBSCRIPTION mysub CONNECTION 'dbname=foo host=bar user=repuser' PUBLICATION
 mypub;
```

Показанная выше команда запустит процесс репликации, который вначале синхронизирует исходное содержимого таблиц `users` и `departments`, а затем начнёт перенос инкрементальных изменений в этих таблицах.

Часть IV. Клиентские интерфейсы

В этой части документации описываются клиентские программные интерфейсы, включённые в дистрибутив Postgres Pro. Все включённые в неё главы можно читать по отдельности. Заметьте, что существует множество других программных интерфейсов, со своей документацией (некоторые наиболее популярные перечислены в Приложении H). Для изучения этой части нужно уметь работать с базой данных, используя команды SQL (см. Часть II), и, конечно же, знать язык программирования, на который ориентирован определённый интерфейс.

Глава 31. libpq — библиотека для языка C

libpq — это интерфейс Postgres Pro для программирования приложений на языке C. Библиотека libpq содержит набор функций, используя которые клиентские программы могут передавать запросы серверу Postgres Pro и принимать результаты этих запросов.

libpq также является базовым механизмом для нескольких других прикладных интерфейсов Postgres Pro, включая те, что написаны для C++, Perl, Python, Tcl и ECPG. Поэтому некоторые аспекты поведения libpq будут важны для вас, если вы используете один из этих пакетов. В частности, Раздел 31.14, Раздел 31.15 и Раздел 31.18 описывают поведение, видимое пользователю любого приложения, использующего libpq.

В конце этой главы приведены короткие программы (Раздел 31.21), показывающие, как использовать libpq в своих программах. В каталоге `src/test/examples` дистрибутивного комплекта исходных текстов приведено несколько завершённых примеров приложений libpq.

Клиентские программы, которые используют libpq, должны включать заголовочный файл `libpq-fe.h` и должны компоноваться с библиотекой libpq.

31.1. Функции управления подключением к базе данных

Следующие функции имеют дело с созданием подключения к серверу Postgres Pro. Прикладная программа может иметь несколько подключений к серверу, открытых одновременно. (Одна из причин этого заключается в необходимости доступа к более чем одной базе данных.) Каждое соединение представляется объектом `PGconn`, который можно получить от функций `PQconnectdb`, `PQconnectdbParams` или `PQsetdbLogin`. Обратите внимание, что эти функции всегда возвратят ненулевой указатель на объект, если только, возможно, не осталось слишком мало памяти даже для того, чтобы выделить её для объекта `PGconn`. Прежде чем передавать запросы через объект подключения, следует вызвать функцию `PQstatus` для проверки возвращаемого значения в случае успешного подключения.

> **Предупреждение**
>
> Если к базе данных, которая не приведена в соответствие шаблону безопасного использования схем, имеют доступ недоверенные пользователи, начинайте сеанс с удаления доступных им для записи схем из пути поиска (`search_path`). Для этого можно присвоить параметру с ключом `options` значение `-csearch_path=`. Также можно выполнить `PQexec(`*соединение*`, "SELECT pg_catalog.set_config('search_path', '', false)")` после подключения. Это касается не только psql, но и любых других интерфейсов для выполнения произвольных SQL-команд.

> **Предупреждение**
>
> В системе Unix создание дочернего процесса на основе процесса, уже имеющего открытые подключения с помощью libpq, может привести к непредсказуемым результатам, потому что родительский и дочерний процессы совместно используют одни и те же сокеты и ресурсы операционной системы. По этой причине подобный подход не рекомендуется. Однако использование системного вызова `exec` из дочернего процесса для загрузки нового исполняемого файла является безопасным.

`PQconnectdbParams`

Создаёт новое подключение к серверу баз данных.

```
PGconn *PQconnectdbParams(const char * const *keywords,
                          const char * const *values,
                          int expand_dbname);
```

Эта функция открывает новое соединение с базой данных, используя параметры, содержащиеся в двух массивах, завершающихся символом `NULL`. Первый из них, `keywords`, определяется как массив строк, каждая из которых представляет собой ключевое слово. Второй, `values`, даёт значение для каждого ключевого слова. В отличие от `PQsetdbLogin`, описываемой ниже, набор параметров может быть расширен без изменения сигнатуры функции, поэтому использование данной функции (или её неблокирующих аналогов `PQconnectStartParams` и `PQconnectPoll`) является предпочтительным при разработке новых приложений.

Ключевые слова-параметры, распознаваемые в настоящее время, приведены в Подразделе 31.1.2.

Передаваемые массивы могут быть пустыми (в этом случае будут использоваться все параметры по умолчанию) или содержать одно или несколько имён/значений параметров. При этом они должны быть одинаковой длины. Просмотр массивов завершается, как только в массиве `keywords` встречается `NULL`. Если элемент массива `values`, соответствующий отличному от `NULL` элементу `keywords`, содержит пустую строку или NULL, такой параметр пропускается и просматривается следующая пара элементов массива.

Когда `expand_dbname` имеет ненулевое значение, первый параметр *`dbname`* может содержать *строку подключения*. В этом случае она «разворачивается» в отдельные параметры подключения, извлечённые из этой строки. Значение считается строкой подключения, а не просто именем базы данных, если оно содержит знак равно (=) или начинается с обозначения схемы URI. (Подробнее форматы строк подключения описаны в Подразделе 31.1.1.) Таким способом обрабатывается только первое вхождение *`dbname`*, следующие параметры *`dbname`* будут восприниматься как просто имя базы данных.

Как правило, массивы параметров обрабатываются от начала к концу. Если какой-либо параметр указывается неоднократно, использоваться будет последнее значение (отличное от `NULL` и непустое). Это справедливо, в частности, и тогда, когда ключевое слово, заданное в строке подключения, конфликтует с заданным в массиве `keywords`. Таким образом, программист может по своему усмотрению решить, будут ли значения в массиве переопределять значения, заданными в строке подключения, или переопределяться ими. Элементы массива, предшествующие развёрнутому значению *`dbname`*, могут быть переопределены значениями в строке подключения, которые в свою очередь переопределяются элементами массива, следующими после *`dbname`* (и в этом случае речь идёт о непустых значениях).

После разбора всех элементов массива и развёрнутой строки подключения (если она задана), параметры подключения, которые остались незаданными, получают значения по умолчанию. Если незаданному параметру соответствует установленная переменная окружения (см. Раздел 31.14), будет использоваться её значение. Если такая переменная не задана, для параметра будет использоваться встроенное значение по умолчанию.

`PQconnectdb`

Создаёт новое подключение к серверу баз данных.

```
PGconn *PQconnectdb(const char *conninfo);
```

Эта функция открывает новое соединение с базой данных, используя параметры, полученные из строки `conninfo`.

Передаваемая строка может быть пустой. В этом случае используются все параметры по умолчанию. Она также может содержать одно или более значений параметров, разделённых пробелами, или URI. За подробностями обратитесь к Подразделу 31.1.1.

`PQsetdbLogin`

Создаёт новое подключение к серверу баз данных.

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,
                     const char *dbName,
                     const char *login,
                     const char *pwd);
```

Это предшественница функции `PQconnectdb` с фиксированным набором параметров. Она имеет такую же функциональность, за исключением того, что непереданные параметры всегда принимают значения по умолчанию. Подставьте `NULL` или пустую строку в качестве любого из фиксированных параметров, которые должны принять значения по умолчанию.

Если параметр *`dbName`* содержит знак = или имеет допустимый префикс URI для подключения, то он воспринимается в качестве строки *`conninfo`* точно таким же образом, как если бы он был передан функции `PQconnectdb`, а оставшиеся параметры затем применяются, как указано для `PQconnectdbParams`.

`PQsetdb`

Создаёт новое подключение к серверу баз данных.

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName);
```

Это макрос, который вызывает `PQsetdbLogin` с нулевыми указателями в качестве значений параметров *`login`* и *`pwd`*. Обеспечивает обратную совместимость с очень старыми программами.

`PQconnectStartParams`
`PQconnectStart`
`PQconnectPoll`

Создают подключение к серверу баз данных неблокирующим способом.

```
PGconn *PQconnectStartParams(const char * const *keywords,
                             const char * const *values,
                             int expand_dbname);

PGconn *PQconnectStart(const char *conninfo);

PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

Три эти функции используются для того, чтобы открыть подключение к серверу баз данных таким образом, чтобы поток исполнения вашего приложения не был заблокирован при выполнении удалённой операции ввода/вывода в процессе подключения. Суть этого подхода в том, чтобы ожидание завершения операций ввода/вывода могло происходить в главном цикле приложения, а не в внутри функций `PQconnectdbParams` или `PQconnectdb`, с тем, чтобы приложение могло управлять этой операцией параллельно с другой работой.

С помощью функции `PQconnectStartParams` подключение к базе данных выполняется, используя параметры, взятые из массивов `keywords` и `values`, а управление осуществляется с помощью `expand_dbname`, как описано выше для `PQconnectdbParams`.

С помощью функции `PQconnectStart` подключение к базе данных выполняется, используя параметры, взятые из строки `conninfo`, как описано выше для `PQconnectdb`.

Ни PQconnectStartParams, ни PQconnectStart, ни PQconnectPoll не заблокируются до тех пор, пока выполняется ряд ограничений:

- Параметр hostaddr должен использоваться так, чтобы для разрешения заданного имени не требовалось выполнять запросы DNS. Подробнее этот параметр описан в Подразделе 31.1.2.
- Если вы вызываете PQtrace, обеспечьте, чтобы поток, в который выводится трассировочная информация, не заблокировался.
- Перед вызовом PQconnectPoll вы должны перевести сокет в соответствующее состояние, как описано ниже.

Чтобы начать неблокирующий запрос на подключение, вызовите PQconnectStart или PQconnectStartParams. Если результатом будет null, значит libpq не смогла выделить память для новой структуры PGconn. В противном случае возвращается действительный указатель PGconn (хотя он ещё не представляет установленное подключение к базе данных). Затем вызовите PQstatus(conn). Если результатом будет CONNECTION_BAD, значит попытка подключения уже не будет успешной, возможно, из-за неверных параметров.

Если вызов PQconnectStart или PQconnectStartParams оказался успешным, теперь нужно опросить libpq для продолжения процедуры подключения. Вызовите PQsocket(conn) для получения дескриптора нижележащего сокета, через который устанавливается соединение. (Внимание: этот сокет может меняться от вызова к вызову PQconnectPoll.) Организуйте цикл таким образом: если PQconnectPoll(conn) при последнем вызове возвращает PGRES_POLLING_READING, ожидайте, пока сокет не окажется готовым для чтения (это покажет функция select(), poll() или подобная системная функция). Затем снова вызовите PQconnectPoll(conn). Если же PQconnectPoll(conn) при последнем вызове возвратила PGRES_POLLING_WRITING, дождитесь готовности сокета к записи, а затем снова вызовите PQconnectPoll(conn). На первой итерации, то есть когда вы ещё не вызывали PQconnectPoll, реализуйте то же поведение, что и после получения PGRES_POLLING_WRITING. Продолжайте этот цикл, пока PQconnectPoll(conn) не выдаст значение PGRES_POLLING_FAILED, сигнализирующее об ошибке при установлении соединения, или PGRES_POLLING_OK, показывающее, что соединение установлено успешно.

В любое время в процессе подключения его состояние можно проверить, вызвав PQstatus. Если этот вызов возвратит CONNECTION_BAD, значит, процедура подключения завершилась сбоем; если вызов возвратит CONNECTION_OK, значит, соединение готово. Оба эти состояния можно определить на основе возвращаемого значения функции PQconnectPoll, описанной выше. Другие состояния могут также иметь место в течение (и только в течение) асинхронной процедуры подключения. Они показывают текущую стадию процедуры подключения и могут быть полезны, например, для предоставления обратной связи пользователю. Вот эти состояния:

CONNECTION_STARTED

Ожидание, пока соединение будет установлено.

CONNECTION_MADE

Соединение установлено; ожидание отправки.

CONNECTION_AWAITING_RESPONSE

Ожидание ответа от сервера.

CONNECTION_AUTH_OK

Аутентификация получена; ожидание завершения запуска серверной части.

CONNECTION_SSL_STARTUP

Согласование SSL-шифрования.

`CONNECTION_SETENV`

Согласование значений параметров, зависящих от программной среды.

`CONNECTION_CHECK_WRITABLE`

Проверка, можно ли через подключение выполнять пишущие транзакции.

`CONNECTION_CONSUME`

Прочтение всех оставшихся ответных сообщений через подключение.

Заметьте, что, хотя эти константы и сохранятся (для поддержания совместимости), приложение никогда не должно полагаться на то, что они появятся в каком-то конкретном порядке или вообще появятся, а также на то, что состояние всегда примет одно из этих документированных значений. Приложение может сделать что-то наподобие:

```
switch(PQstatus(conn))
{
        case CONNECTION_STARTED:
            feedback = "Подключение...";
            break;

        case CONNECTION_MADE:
            feedback = "Подключён к серверу...";
            break;
.
.
.
        default:
            feedback = "Подключение...";
}
```

Параметр подключения `connect_timeout` игнорируется, когда используется `PQconnectPoll`; именно приложение отвечает за принятие решения о том, является ли истекшее время чрезмерным. В противном случае вызов `PQconnectStart` с последующим вызовом `PQconnectPoll в цикле` будут эквивалентны вызову `PQconnectdb`.

Заметьте, что если функция `PQconnectStart` или `PQconnectStartParams` возвращает ненулевой указатель, вы должны вызвать `PQfinish`, закончив его использование, чтобы освободить полученную структуру и все связанные с ней блоки памяти. Это нужно сделать, даже если попытка подключения не последует или окажется неуспешной.

`PQconndefaults`

Возвращает значения по умолчанию для параметров подключения.

```
PQconninfoOption *PQconndefaults(void);

typedef struct
{
    char   *keyword;   /* Ключевое слово для данного параметра */
    char   *envvar;    /* Имя альтернативной переменной окружения */
    char   *compiled;  /* Альтернативное значение по умолчанию, назначенное при
 компиляции */
    char   *val;       /* Текущее значение параметра или NULL */
    char   *label;     /* Обозначение этого поля в диалоге подключения */
    char   *dispchar;  /* Показывает, как отображать это поле
                          в диалоге подключения. Значения следующие:
                          ""        Отображать введённое значение "как есть"
                          "*"       Поле пароля — скрывать значение
                          "D"       Параметр отладки — не показывать по умолчанию */
```

```
    int     dispsize;  /* Размер поля в символах для диалога */
} PQconninfoOption;
```

Возвращает массив параметров подключения. Он может использоваться для определения всех возможных параметров `PQconnectdb` и их текущих значений по умолчанию. Возвращаемое значение указывает на массив структур `PQconninfoOption`, который завершается элементом, имеющим нулевой указатель `keyword`. Если выделить память не удалось, то возвращается нулевой указатель. Обратите внимание, что текущие значения по умолчанию (поля `val`) будут зависеть от переменных среды и другого контекста. Отсутствующий или неверный сервисный файл будет молча проигнорирован. Вызывающие функции должны рассматривать данные параметров по умолчанию как «только для чтения».

После обработки массива параметров освободите память, передав его функции `PQconninfoFree`. Если этого не делать, то при каждом вызове функции `PQconndefaults` будет происходить небольшая утечка памяти.

`PQconninfo`

Возвращает параметры подключения, используемые действующим соединением.

```
PQconninfoOption *PQconninfo(PGconn *conn);
```

Возвращает массив параметров подключения. Он может использоваться для определения всех возможных параметров `PQconnectdb` и значений, которые были использованы для подключения к серверу. Возвращаемое значение указывает на массив структур `PQconninfoOption`, который завершается элементом, имеющим нулевой указатель `keyword`. Все замечания, приведённые выше для `PQconndefaults`, также справедливы и для результата `PQconninfo`.

`PQconninfoParse`

Возвращает разобранные параметры подключения, переданные в строке подключения.

```
PQconninfoOption *PQconninfoParse(const char *conninfo, char **errmsg);
```

Разбирает строку подключения и возвращает результирующие параметры в виде массива; возвращает `NULL`, если возникают проблемы при разборе строки подключения. Эту функцию можно использовать для извлечения параметров функции `PQconnectdb` из предоставленной строки подключения. Возвращаемое значение указывает на массив структур `PQconninfoOption`, который завершается элементом, имеющим нулевой указатель `keyword`.

Все разрешённые параметры будут присутствовать в результирующем массиве, но `PQconninfoOption` для любого параметра, не присутствующего в строке подключения, будет иметь значение `NULL` в поле `val`; значения по умолчанию не подставляются.

Если `errmsg` не равно `NULL`, тогда в случае успеха `*errmsg` присваивается `NULL`, а в противном случае -- адрес строки сообщения об ошибке, объясняющего проблему. Память для этой строки выделяет функция `malloc`. (Также возможна ситуация, когда `*errmsg` будет установлено в `NULL`, и при этом функция возвращает `NULL`. Это указывает на нехватку памяти.)

После обработки массива параметров освободите память, передав его функции `PQconninfoFree`. Если этого не делать, тогда некоторое количество памяти будет утекать при каждом вызове `PQconninfoParse`. И наоборот, если произошла ошибка и `errmsg` не равно `NULL`, обязательно освободите память, занимаемую строкой сообщения об ошибке, используя `PQfreemem`.

`PQfinish`

Закрывает соединение с сервером. Также освобождает память, используемую объектом `PGconn`.

```
void PQfinish(PGconn *conn);
```

Обратите внимание, что даже если попытка подключения к серверу потерпела неудачу (как показывает `PQstatus`), приложение все равно должно вызвать `PQfinish`, чтобы освободить

память, используемую объектом PGconn. Указатель PGconn не должен использоваться повторно после того, как была вызвана функция PQfinish.

PQreset

Переустанавливает канал связи с сервером.

```
void PQreset(PGconn *conn);
```

Эта функция закроет подключение к серверу, а потом попытается установить новое подключение, используя все те же параметры, которые использовались прежде. Это может быть полезным для восстановления после ошибки, если работающее соединение было разорвано.

PQresetStart
PQresetPoll

Переустанавливает канал связи с сервером неблокирующим способом.

```
int PQresetStart(PGconn *conn);

PostgresPollingStatusType PQresetPoll(PGconn *conn);
```

Эти функции закроют подключение к серверу, а потом попытаются установить новое подключение, используя все те же параметры, которые использовались прежде. Это может быть полезным для восстановления после ошибки, если работающее соединение оказалось потерянным. Они отличаются от PQreset (см. выше) тем, что действуют неблокирующим способом. На эти функции налагаются те же ограничения, что и на PQconnectStartParams, PQconnectStart и PQconnectPoll.

Чтобы приступить к переустановке подключения, вызовите PQresetStart. Если она возвратит 0, переустановка завершилась неудачно. Если она возвратит 1, опросите результат переустановки, используя PQresetPoll, точно таким же образом, как если бы вы создавали подключение, используя PQconnectPoll.

PQpingParams

PQpingParams сообщает состояние сервера. Она принимает параметры подключения, идентичные тем, что получает функция PQconnectdbParams, описанная выше. Нет необходимости предоставлять корректные имя пользователя, пароль или имя базы данных, чтобы получить состояние сервера. Однако если предоставлены некорректные значения, сервер занесёт в журнал неудачную попытку подключения.

```
PGPing PQpingParams(const char * const *keywords,
                    const char * const *values,
                    int expand_dbname);
```

Функция возвращает одно из следующих значений:

PQPING_OK

Сервер работает и, по-видимому, принимает подключения.

PQPING_REJECT

Сервер работает, но находится в состоянии, которое запрещает подключения (запуск, завершение работы или восстановление после аварийного отказа).

PQPING_NO_RESPONSE

Контакт с сервером не удался. Это может указывать на то, что сервер не запущен или что-то не в порядке с параметрами данного подключения (например, неверный номер порта), или имеет место проблема с возможностью соединения по сети (например, брандмауэр блокирует запрос на подключение).

`PQPING_NO_ATTEMPT`

Никакой попытки установить контакт с сервером сделано не было, поскольку предоставленные параметры были явно некорректными, или имела место какая-то проблема на стороне клиента (например, нехватка памяти).

`PQping`

`PQping` сообщает состояние сервера. Она принимает параметры подключения, идентичные тем, что получает функция `PQconnectdb`, описанная выше. Нет необходимости предоставлять корректные имя пользователя, пароль или имя базы данных, чтобы получить состояние сервера. Однако если предоставлены некорректные значения, сервер занесёт в журнал неудачную попытку подключения.

```
PGPing PQping(const char *conninfo);
```

Возвращаемые значения такие же, как и для `PQpingParams`.

31.1.1. Строки параметров подключения

Ряд функций libpq разбирают заданную пользователем строку для извлечения параметров подключения. Есть два принятых формата этих строк: простая строка ключ/значение и URI. Строки URI в основном соответствуют *RFC 3986*, но могут содержать также строки подключения с несколькими узлами, как описано ниже.

31.1.1.1. Строки параметров подключения вида «ключ/значение»

Согласно формату ключ/значение, установка каждого параметра выполняется в форме *`ключ = значение`*, с пробелами между параметрами. Пробелы вокруг знака равенства не являются обязательными. Для записи пустого значения или значения, содержащего пробелы, заключите его в одинарные кавычки, например, `keyword = 'a value'`. Одинарные кавычки и символы обратной косой черты внутри значения нужно обязательно экранировать с помощью символа обратной косой черты, т. е., `\'` и `\\`.

Пример:

```
host=localhost port=5432 dbname=mydb connect_timeout=10
```

Ключевые слова-параметры, распознаваемые в настоящее время, приведены в Подразделе 31.1.2.

31.1.1.2. URI для подключения

Основная форма URI подключения:

```
postgresql://[пользователь@][сервер][/база_данных][?указание_параметра]

где пользователь:

имя_пользователя[:пароль]

и сервер:

[узел][:порт][,...]

и указание_параметра:

имя=значение[&...]
```

В качестве обозначения схемы URI может использоваться `postgresql://` или `postgres://`. Остальные части URI являются необязательными. В следующих примерах показан допустимый синтаксис URI:

```
postgresql://
postgresql://localhost
```

```
postgresql://localhost:5433
postgresql://localhost/mydb
postgresql://user@localhost
postgresql://user:secret@localhost
postgresql://other@localhost/otherdb?connect_timeout=10&application_name=myapp
postgresql://host1:123,host2:456/somedb?target_session_attrs=any&application_name=myapp
```

Значения, которые обычно задаются в иерархической части URI, также можно задать в именованных параметрах. Например:

```
postgresql:///mydb?host=localhost&port=5433
```

Все именованные параметры должны соответствовать ключевым словам, перечисленным в Подраздел 31.1.2, за исключением того, что вхождения `ssl=true` заменяются на `sslmode=require` для совместимости с URI-строками JDBC.

Для включения символов, имеющих специальное значение, в любой части URI можно применять URL-кодирование (с использованием %), например заменить = на `%3D`.

Сервер можно представить либо сетевым именем, либо IP-адресом. При использовании протокола IPv6 нужно заключить адрес в квадратные скобки:

```
postgresql://[2001:db8::1234]/database
```

Часть `host` интерпретируется в соответствии с описанием параметра host. Так, если эта часть строки пуста или выглядит как абсолютный путь, выбирается соединение через Unix-сокеты, в противном случае инициируется соединение по TCP/IP. Учтите однако, что символ косой черты в иерархической части URI является зарезервированным. Поэтому, чтобы указать нестандартный каталог Unix-сокета, нужно поступить одним из двух способов: не задавать эту часть в URI и указать сервер в именованном параметре либо закодировать путь в части `host` с процентами:

```
postgresql:///dbname?host=/var/lib/postgresql
postgresql://%2Fvar%2Flib%2Fpostgresql/dbname
```

В одном URI можно задать несколько компонентов узлов, каждый с необязательным указанием порта. URI вида `postgresql://host1:port1,host2:port2,host3:port3/` равнозначно строке подключения вида `host=host1,host2,host3 port=port1,port2,port3`. Как описывается ниже, эти узлы будут перебираться по очереди, пока не будет установлено подключение.

31.1.1.3. Указание нескольких узлов

В строке подключения можно задать несколько узлов, к которым клиент будет пытаться подключиться в заданном порядке. Параметры `host`, `hostaddr` и `port` в формате ключ/значение принимают списки значений, разделённых запятыми. В каждом определяемом параметре должно содержаться одинаковое число элементов, чтобы, например, первый элемент `hostaddr` соответствовал первому элементу `host`, второй — второму `host` и так далее. Исключение составляет `port` — если этот параметр содержит только один элемент, он применяется ко всем узлам.

В формате URI внутри компонента `host` можно указать несколько пар `host:port`, разделённых запятыми.

В любом формате одно имя узла может переводиться в несколько сетевых адресов. Например, часто бывает, что один узел имеет и адрес IPv4, и адрес IPv6.

Когда задаются несколько узлов или когда одно имя узла переводится в несколько адресов, все узлы и адреса перебираются по порядку, пока подключение не будет установлено. Если ни один из адресов не будет доступен, произойдёт сбой подключения. Если подключение устанавливается успешно, но происходит ошибка аутентификации, остальные узлы в списке не перебираются.

Если используется файл паролей, в нём можно задать разные пароли для разных узлов. Все остальные параметры подключения будут одинаковыми для всех узлов; например, нельзя задать для разных узлов различные имена пользователей.

31.1.2. Ключевые слова-параметры

Ключевые слова-параметры, распознаваемые в настоящее время, следующие:

`host`

Имя компьютера для подключения. Если это имя начинается с косой черты, оно выбирает подключение через Unix-сокет, а не через TCP/IP, и задаёт имя каталога, содержащего файл сокета. По умолчанию, если параметр `host` отсутствует или пуст, выполняется подключение к Unix-сокету в `/tmp` (или в том каталоге сокетов, который был задан при сборке Postgres Pro). В системах, где Unix-сокеты не поддерживаются, по умолчанию выполняется подключение к `localhost`.

Также принимается разделённый запятыми список имён узлов; при этом данные имена будут перебираться по порядку. Для пустых элементов списка применяется поведение по умолчанию, описанное выше. За подробностями обратитесь к Подразделу 31.1.1.3.

`hostaddr`

Числовой IP-адрес компьютера для подключения. Он должен быть представлен в стандартном формате адресов IPv4, например, `172.28.40.9`. Если ваша машина поддерживает IPv6, вы можете использовать и эти адреса. Связь по протоколу TCP/IP используется всегда, когда в качестве этого параметра передана непустая строка.

Использование `hostaddr` вместо `host` позволяет приложению избежать поиска на сервере имён, что может быть важно для приложений, имеющих ограничения по времени. Однако имя компьютера требуется для методов аутентификации GSSAPI или SSPI, а также для проверки полномочий на основе SSL-сертификатов в режиме `verify-full`. Применяются следующие правила:

- Если адрес `host` задаётся без `hostaddr`, осуществляется разрешение имени. (При использовании `PQconnectPoll` разрешение производится, когда `PQconnectPoll` рассматривает это имя в первый раз, и может заблокировать `PQconnectPoll` на неопределённое время.)
- Если указан `hostaddr`, а `host` не указан, тогда значение `hostaddr` даёт сетевой адрес сервера. Попытка подключения завершится неудачей, если метод аутентификации требует наличия имени компьютера.
- Если указаны как `host`, так и `hostaddr`, тогда значение `hostaddr` даёт сетевой адрес сервера, а значение `host` игнорируется, если только метод аутентификации его не потребует. В таком случае оно будет использоваться в качестве имени компьютера.

Заметьте, что аутентификация может завершится неудачей, если `host` не является именем сервера с сетевым адресом `hostaddr`. Также заметьте, что когда указывается и `host`, и `hostaddr`, соединение в файле паролей идентифицируется по значению `host` (см. Раздел 31.15).

Также принимается разделённый запятыми список значений `hostaddr`, при этом данные узлы будут перебираться по порядку. Вместо пустого элемента в этом списке будет подставлено имя соответствующего узла или, если и оно не определено, имя узла по умолчанию. За подробностями обратитесь к Подразделу 31.1.1.3.

Если не указаны ни имя компьютера, ни его адрес, libpq будет производить подключение, используя локальный Unix-сокет; в системах, не поддерживающих Unix-сокеты, она будет пытаться подключиться к `localhost`.

`port`

Номер порта, к которому нужно подключаться на сервере, либо расширение имени файла сокета для соединений через Unix-сокеты. Если в параметрах `host` или `hostaddr` задано несколько серверов, в данном параметре может задаваться через запятую список портов такой же длины, либо может указываться один номер порта для всех узлов. Пустая строка

или пустой элемент в списке через запятую воспринимается как номер порта по умолчанию, установленный при сборке PostgreSQL.

`dbname`

Имя базы данных. По умолчанию оно совпадает с именем пользователя. В определённых контекстах это значение проверяется на соответствие расширенным форматам; см. Подраздел 31.1.1 для получения подробной информации.

`user`

Имя пользователя Postgres Pro, используемое для подключения. По умолчанию используется то же имя, которое имеет в операционной системе пользователь, от лица которого выполняется приложение.

`password`

Пароль, используемый в случае, когда сервер требует аутентификации по паролю.

`passfile`

Задаёт имя файла, в котором будут храниться пароли (см. Раздел 31.15). По умолчанию это `~/.pgpass` или `%APPDATA%\postgresql\pgpass.conf` в Microsoft Windows. (Отсутствие этого файла не считается ошибкой.)

`connect_timeout`

Максимальное время ожидания подключения (задаётся десятичным целым числом, например: `10`). При нуле, отрицательном или неопределённом значении ожидание будет бесконечным. Минимальный допустимый тайм-аут равен 2 секундам; таким образом, значение `1` воспринимается как `2`. Этот тайм-аут применяется для каждого отдельного IP-адреса или имени сервера. Например, если вы зададите адреса двух серверов и значение `connect_timeout` будет равно 5, тайм-аут при неудачной попытке подключения к каждому серверу произойдёт через 5 секунд, а общее время ожидания подключения может достигать 10 секунд.

`client_encoding`

Этим устанавливается конфигурационный параметр `client_encoding` для данного подключения. В дополнение к значениям, которые принимает соответствующий параметр сервера, вы можете использовать значение `auto`. В этом случае правильная кодировка определяется на основе текущей локали на стороне клиента (в системах Unix это переменная системного окружения `LC_CTYPE`).

`options`

Задаёт параметры командной строки, которые будут отправлены серверу при установлении соединения. Например, значение `-c geqo=off` установит для параметра сеанса `geqo` значение `off`. Пробелы в этой строке считаются разделяющими аргументы командной строки, если только перед ними не стоит обратная косая черта (`\`); чтобы записать собственно обратную косую черту, её нужно продублировать (`\\`). Подробное описание возможных параметров можно найти в Главе 18.

`application_name`

Устанавливает значение для конфигурационного параметра application_name.

`fallback_application_name`

Устанавливает альтернативное значение для конфигурационного параметра application_name. Это значение будет использоваться, если для параметра `application_name` не было передано никакого значения с помощью параметров подключения или переменной системного окружения `PGAPPNAME`. Задание альтернативного имени полезно для универсальных программ-утилит, которые желают установить имя приложения по умолчанию, но позволяют пользователю изменить его.

`keepalives`

Управляет использованием сообщений keepalive протокола TCP на стороне клиента. Значение по умолчанию равно 1, что означает использование сообщений. Вы можете изменить его на 0, если эти сообщения не нужны. Для соединений, установленных через Unix-сокеты, этот параметр игнорируется.

`keepalives_idle`

Управляет длительностью периода отсутствия активности, выраженного числом секунд, по истечении которого TCP должен отправить сообщение keepalive серверу. При значении 0 действует системная величина. Этот параметр игнорируется для соединений, установленных через Unix-сокеты, или если сообщения keepalive отключены. Он поддерживается только в системах, воспринимающих параметр сокета `TCP_KEEPIDLE` или равнозначный, и в Windows; в других системах он не оказывает влияния.

`keepalives_interval`

Задаёт количество секунд, по прошествии которых сообщение keepalive протокола TCP, получение которого не подтверждено сервером, должно быть отправлено повторно. При значении 0 действует системная величина. Этот параметр игнорируется для соединений, установленных через Unix-сокеты, или если сообщения keepalive отключены. Он поддерживается только в системах, воспринимающих параметр сокета `TCP_KEEPINTVL` или равнозначный, и в Windows; в других системах он не оказывает влияния.

`keepalives_count`

Задаёт количество сообщений keepalive протокола TCP, которые могут быть потеряны, прежде чем соединение клиента с сервером будет признано неработающим. Нулевое значение этого параметра указывает, что будет использоваться системное значение по умолчанию. Этот параметр игнорируется для соединений, установленных через Unix-сокеты, или если сообщения keepalive отключены. Он поддерживается только в системах, воспринимающих параметр сокета `TCP_KEEPCNT` или равнозначный; в других системах он не оказывает влияния.

`tty`

Игнорируется (прежде он указывал, куда направить вывод отладочных сообщений сервера).

`sslmode`

Этот параметр определяет, будет ли согласовываться с сервером защищённое SSL-соединение по протоколу TCP/IP, и если да, то в какой очередности. Всего предусмотрено шесть режимов:

`disable`

следует пытаться установить только соединение без использования SSL

`allow`

сначала следует попытаться установить соединение без использования SSL; если попытка будет неудачной, нужно попытаться установить SSL-соединение

`prefer` (по умолчанию)

сначала следует попытаться установить SSL-соединение; если попытка будет неудачной, нужно попытаться установить соединение без использования SSL

`require`

следует попытаться установить только SSL-соединение. Если присутствует файл корневого центра сертификации, то нужно верифицировать сертификат таким же способом, как будто был указан параметр `verify-ca`

`verify-ca`

следует попытаться установить только SSL-соединение, при этом проконтролировать, чтобы сертификат сервера был выпущен доверенным центром сертификации (CA)

`verify-full`

следует попытаться установить только SSL-соединение, при этом проконтролировать, чтобы сертификат сервера был выпущен доверенным центром сертификации (CA) и чтобы имя запрошенного сервера соответствовало имени в сертификате

В Разделе 31.18 приведено подробное описание работы этих режимов.

`sslmode` игнорируется при использовании Unix-сокетов. Если Postgres Pro скомпилирован без поддержки SSL, использование параметров `require`, `verify-ca` или `verify-full` приведёт к ошибке, в то время как параметры `allow` и `prefer` будут приняты, но libpq в действительности не будет пытаться установить SSL-соединение.

`requiressl`

Использовать этот параметр не рекомендуется, в качестве замены предлагается установить `sslmode`.

Если установлено значение 1, то требуется SSL-соединение с сервером (это эквивалентно `sslmode require`). libpq в таком случае откажется подключаться, если сервер не принимает SSL-соединений. Если установлено значение 0 (по умолчанию), тогда libpq будет согласовывать тип подключения с сервером (эквивалентно `sslmode prefer`). Этот параметр доступен, если только Postgres Pro скомпилирован с поддержкой SSL.

`sslcompression`

Если установлено значение 1 (по умолчанию), данные, пересылаемые через SSL-соединения, будут сжиматься. Если установлено значение 0, сжатие будет отключено (для этого требуется OpenSSL версии 1.0.0 или более поздней). Этот параметр игнорируется, если выполнено подключение без SSL, или если используемая версия OpenSSL не поддерживает его.

Сжатие требует процессорного времени, но может улучшить пропускную способность, если узким местом является сеть. Отключение сжатия может улучшить время отклика и пропускную способность, если ограничивающим фактором является производительность CPU.

`sslcert`

Этот параметр предписывает имя файла для SSL-сертификата клиента, заменяющего файл по умолчанию `~/.postgresql/postgresql.crt`. Этот параметр игнорируется, если SSL-подключение не выполнено.

`sslkey`

Этот параметр предписывает местоположение секретного ключа, используемого для сертификата клиента. Он может либо указывать имя файла, которое будет использоваться вместо имени по умолчанию `~/.postgresql/postgresql.key`, либо он может указывать ключ, полученный от внешнего «криптомодуля» (криптомодули — это загружаемые модули OpenSSL). Спецификация внешнего криптомодуля должна состоять из имени модуля и ключевого идентификатора, зависящего от конкретного модуля, разделённых двоеточием. Этот параметр игнорируется, если SSL-подключение не выполнено.

`sslrootcert`

Этот параметр указывает имя файла, содержащего SSL-сертификаты, выданные Центром сертификации (CA). Если файл существует, сертификат сервера будет проверен на предмет его подписания одним из этих центров. Имя по умолчанию — `~/.postgresql/root.crt`.

`sslcrl`

Этот параметр указывает имя файла, содержащего список отозванных серверных сертификатов (CRL) для SSL. Сертификаты, перечисленные в этом файле, если он существует, будут отвергаться при попытке установить подлинность сертификата сервера. Имя по умолчанию такое `~/.postgresql/root.crl`.

`requirepeer`

Этот параметр указывает имя пользователя операционной системы, предназначенное для сервера, например, `requirepeer=postgres`. При создании подключения через Unix-сокет, если этот параметр установлен, клиент проверяет в самом начале процедуры подключения, что серверный процесс запущен от имени указанного пользователя; если это не так, соединение аварийно прерывается с ошибкой. Этот параметр можно использовать, чтобы обеспечить аутентификацию сервера, подобную той, которая доступна с помощью SSL-сертификатов при соединениях по протоколу TCP/IP. (Заметьте, что если Unix-сокет находится в каталоге `/tmp` или в другом каталоге, запись в который разрешена всем пользователям, тогда любой пользователь сможет запустить сервер, прослушивающий сокет в том каталоге. Используйте этот параметр, чтобы гарантировать, что вы подключены к серверу, запущенному доверенным пользователем.) Он поддерживается только на платформах, для которых реализован метод аутентификации `peer`; см. Подраздел 19.3.6.

`krbsrvname`

Имя сервиса Kerberos, предназначенное для использования при аутентификации на основе GSSAPI. Оно должно соответствовать имени сервиса, указанному в конфигурации сервера, чтобы аутентификация на основе Kerberos прошла успешно. (См. также Подраздел 19.3.3.)

`gsslib`

Библиотека GSS, предназначенная для использования при аутентификации на основе GSSAPI. В настоящее время это действует только в сборках для Windows, поддерживающих одновременно и GSSAPI, и SSPI. Значение `gssapi` в таких сборках позволяет указать, что libpq должна использовать для аутентификации библиотеку GSSAPI, а не подразумеваемую по умолчанию SSPI.

`service`

Имя сервиса, используемое для задания дополнительных параметров. Оно указывает имя сервиса в файле `pg_service.conf`, который содержит дополнительные параметры подключения. Это позволяет приложениям указывать только имя сервиса, поскольку параметры подключения могут поддерживаться централизованно. См. Раздел 31.16.

`target_session_attrs`

Если этот параметр равен `read-write`, по умолчанию будут приемлемы только подключения, допускающие транзакции на чтение/запись. При успешном подключении будет отправлен запрос `SHOW transaction_read_only`; если он вернёт `on`, соединение будет закрыто. Если в строке подключения указано несколько серверов, будут перебираться остальные серверы, как и при неудачной попытке подключения. Со значением по умолчанию (`any`) приемлемыми будут все подключения.

31.2. Функции, описывающие текущее состояние подключения

Эти функции могут использоваться для опроса состояния объекта, описывающего существующее подключение к базе данных.

> **Подсказка**
>
> Разработчики приложений на основе libpq должны тщательно поддерживать абстракцию `PGconn`. Следует использовать функции доступа, описанные ниже, для получения содержимого `PGconn`. Обращение напрямую к внутренним полям `PGconn`, используя сведения из `libpq-int.h`, не рекомендуется, поскольку они могут измениться в будущем.

Следующие функции возвращают значения параметров, которые были установлены в момент подключения. Эти значения не изменяются во время жизни соединения. Если используется строка

соединения с несколькими узлами, значения PQhost, PQport и PQpass могут меняться, если с тем же объектом PGconn устанавливается новое соединение. Другие значения не меняются на протяжении жизни объекта PGconn.

PQdb

Возвращает имя базы данных, с которой установлено соединение.

```
char *PQdb(const PGconn *conn);
```

PQuser

Возвращает имя пользователя, который установил соединение.

```
char *PQuser(const PGconn *conn);
```

PQpass

Возвращает пароль, использованный для подключения.

```
char *PQpass(const PGconn *conn);
```

PQpass возвратит либо пароль, указанный в параметрах подключения, либо пароль, полученный из файла паролей (в случае отсутствия первого). Во втором случае, если в параметрах подключения было указано несколько узлов, полагаться на результат PQpass нельзя, пока соединение не будет установлено. Состояние подключения позволяет проверить функция PQstatus.

PQhost

Возвращает имя сервера для активного соединения. Это может быть имя компьютера, IP-адрес или путь к каталогу, если подключение установлено через сокет Unix. (Признаком подключения к сокету будет абсолютный путь, который начинается с /.)

```
char *PQhost(const PGconn *conn);
```

Если в параметрах подключения был задан и host, и hostaddr, функция PQhost выдаст содержимое host. Если был задан только hostaddr, возвращается это значение. Если в параметрах подключения было задано несколько узлов, PQhost возвращает адрес узла, с которым фактически установлено соединение.

PQhost возвращает NULL, если аргумент *conn* равен NULL. Иначе в случае ошибки при получении информации об узле (это возможно, если соединение не установлено до конца или произошла ошибка) она возвращает пустую строку.

Если в параметрах подключения указаны несколько узлов, на результат PQhost нельзя полагаться, пока соединение не будет установлено. Проверить состояние соединения позволяет функция PQstatus.

PQport

Возвращает номер порта активного соединения.

```
char *PQport(const PGconn *conn);
```

Если в параметрах подключения было задано несколько портов, PQport возвращает порт, с которым фактически установлено соединение.

PQport возвращает NULL, если аргумент *conn* равен NULL. Иначе в случае ошибки при получении информации о порте (это возможно, если соединение не установлено до конца или произошла ошибка) она возвращает пустую строку.

Если в параметрах подключения указаны несколько портов, на результат PQport нельзя полагаться, пока соединение не будет установлено. Проверить состояние соединения позволяет функция PQstatus.

PQtty

Возвращает имя отладочного терминала (TTY), связанного с данным соединением. (Это устаревшая функция, поскольку сервер более не обращает внимания на установку TTY, но она остаётся для обеспечения обратной совместимости.)

```
char *PQtty(const PGconn *conn);
```

PQoptions

Возвращает параметры командной строки, переданные в запросе на подключение.

```
char *PQoptions(const PGconn *conn);
```

Следующие функции возвращают данные статуса, который может измениться в процессе выполнения операций на объекте PGconn.

PQstatus

Возвращает состояние подключения.

```
ConnStatusType PQstatus(const PGconn *conn);
```

Статус может принимать одно из ряда значений. Однако только два из них видны извне процедуры асинхронного подключения: CONNECTION_OK и CONNECTION_BAD. Успешное подключение к базе данных имеет статус CONNECTION_OK. О неудачной попытке подключения сигнализирует статус CONNECTION_BAD. Обычно статус OK остаётся таковым до вызова PQfinish, но сбой в коммуникации может привести к тому, что статус изменится на CONNECTION_BAD преждевременно. В таком случае приложение может попытаться восстановиться, вызвав PQreset.

О других кодах состояния, которые могут выдать эти функции, можно узнать в описании PQconnectStartParams, PQconnectStart и PQconnectPoll.

PQtransactionStatus

Возвращает текущий статус сервера, отражающий процесс выполнения транзакций.

```
PGTransactionStatusType PQtransactionStatus(const PGconn *conn);
```

Статус может быть одним из PQTRANS_IDLE (в настоящее время не занят обработкой транзакции), PQTRANS_ACTIVE (команда в процессе обработки), PQTRANS_INTRANS (не выполняет работу, но находится в рамках действительной транзакции) или PQTRANS_INERROR (не выполняет работу, но находится в рамках транзакции, завершившейся сбоем). Статус принимает значение PQTRANS_UNKNOWN, если соединение не работает. Статус принимает значение PQTRANS_ACTIVE только тогда, когда запрос был отправлен серверу, но ещё не завершён.

PQparameterStatus

Отыскивает текущее значение параметра сервера.

```
const char *PQparameterStatus(const PGconn *conn, const char *paramName);
```

Значения определённых параметров сервер сообщает автоматически в начале процедуры подключения или тогда, когда их значения изменяются. PQparameterStatus можно использовать, чтобы запросить эти значения. Функция возвращает текущее значение параметра, если оно известно, или NULL, если параметр неизвестен.

Параметры, значения которых сообщает сервер, в текущей версии включают server_version, server_encoding, client_encoding, application_name, is_superuser, session_authorization, DateStyle, IntervalStyle, TimeZone, integer_datetimes и standard_conforming_strings. (Значения параметров server_encoding, TimeZone и integer_datetimes сервер до версии 8.0 не сообщал; standard_conforming_strings сервер до версии 8.1 не сообщал; IntervalStyle

сервер до версии 8.4 не сообщал; application_name сервер до версии 9.0 не сообщал.) Учтите, что server_version, server_encoding и integer_datetimes нельзя изменить после запуска.

Серверы, поддерживающие протокол только до версии 3.0, не могут сообщать установки параметров, но libpq включает средства, чтобы получить значения для server_version и client_encoding в любом случае. Поощряется использование в приложениях PQparameterStatus, а не специально написанного (*ad hoc*) кода, для определения этих значений. (Примите к сведению, однако, что в соединениях, основанных на протоколе версии до 3.0, изменение client_encoding посредством команды SET после начала процедуры подключения не будет отражаться функцией PQparameterStatus.) Сведения о server_version приведены также в описании функции PQserverVersion, возвращающей информацию в числовой форме, в которой гораздо легче выполнять сравнение.

Если для standard_conforming_strings не передано никакого значения, то приложения могут принять его равным off. Это означает, что символы обратной косой черты в строковых литералах интерпретируются в качестве спецсимволов. Также, наличие этого параметра может рассматриваться как указание на то, что синтаксис escape-строк (E'...') является приемлемым.

Хотя возвращаемый указатель объявлен со спецификатором const, фактически он указывает на изменяемое хранилище, связанное со структурой PGconn. Не стоит рассчитывать на то, что указатель останется действительным при последующих запросах.

PQprotocolVersion

Запрашивает протокол, используемый между клиентом и сервером.

```
int PQprotocolVersion(const PGconn *conn);
```

Приложения могут использовать эту функцию, чтобы определить, поддерживаются ли определённые функциональные возможности. В настоящее время возможными значениями являются 2 (протокол версии 2.0), 3 (протокол версии 3.0) или ноль (проблемы в подключении). Версия протокола не будет изменяться после завершения процедуры подключения, но теоретически она могла бы измениться в процессе переподключения. Версия протокола 3.0 обычно используется при взаимодействии с серверами PostgreSQL версии 7.4 или более поздними; серверы до версии 7.4 поддерживают только протокол версии 2.0. (Протокол версии 1.0 является устаревшим и не поддерживается библиотекой libpq.)

PQserverVersion

Возвращает целочисленное представление версии сервера.

```
int PQserverVersion(const PGconn *conn);
```

Приложения могут использовать эту функцию, чтобы определить версию сервера баз данных, к которому они подключены. Возвращаемое ей число получается в результате умножения номера основной версии на 10000 и добавления номера дополнительной версии. Например, для версии 10.1 будет выдано число 100001, а для версии 11.0 — 110000. Если соединение не работает, то возвращается ноль.

До версии 10, в Postgres Pro номера версий образовывались из трёх чисел, первые два из которых представляли основную версию. Для таких версий PQserverVersion отводит на каждое число по две цифры; например, для версии 9.1.5 будет выдано 90105, а для версии 9.2.0 — 90200.

Таким образом, чтобы получить логический номер версии для определения доступности функционала, приложения должны разделить результат PQserverVersion на 100, а не на 10000. Во всех сериях номера дополнительных (корректирующих) выпусков различаются только в двух последних цифрах.

PQerrorMessage

Возвращает сообщение об ошибке, наиболее недавно сгенерированное операцией, выполненной в рамках текущего подключения.

```
char *PQerrorMessage(const PGconn *conn);
```

Почти все функции библиотеки libpq в случае сбоя сформируют сообщение для `PQerrorMessage`. Обратите внимание, что по соглашениям, принятым в libpq, непустой результат функции `PQerrorMessage` может состоять из нескольких строк и будет включать завершающий символ новой строки. Вызывающая функция не должна освобождать память, на которую указывает возвращаемое значение, напрямую. Она будет освобождена, когда связанный с ней дескриптор `PGconn` будет передан функции `PQfinish`. Не стоит ожидать, что результирующая строка останется той же самой при выполнении нескольких операций со структурой `PGconn`.

`PQsocket`

Получает номер файлового дескриптора для сокета соединения с сервером. Действительный дескриптор будет больше или равен 0; значение -1 показывает, что в данный момент не открыто ни одного соединения с сервером. (Значение не изменится во время обычной работы, но может измениться во время установки или переустановки подключения.)

```
int PQsocket(const PGconn *conn);
```

`PQbackendPID`

Возвращает ID (PID) серверного процесса, обрабатывающего это подключение.

```
int PQbackendPID(const PGconn *conn);
```

PID серверного процесса полезен для отладочных целей и для сопоставления с сообщениями команды `NOTIFY` (которые включают PID уведомляющего серверного процесса). Примите к сведению, что PID принадлежит процессу, выполняющемуся на компьютере сервера баз данных, а не на локальном компьютере.

`PQconnectionNeedsPassword`

Возвращает true (1), если метод аутентификации соединения требовал пароля, однако он не был предоставлен. Возвращает false (0), если пароль не требовался.

```
int PQconnectionNeedsPassword(const PGconn *conn);
```

Эту функцию можно применить после неудачной попытки подключения, чтобы решить, нужно ли предлагать пользователю ввести пароль.

`PQconnectionUsedPassword`

Возвращает true (1), если метод аутентификации соединения использовал пароль. Возвращает false (0) в противном случае.

```
int PQconnectionUsedPassword(const PGconn *conn);
```

Эту функцию можно применить как после неудачной, так и после успешной попытки подключения, чтобы определить, требовал ли сервер предоставления пароля.

Следующие функции возвращают информацию, относящуюся к SSL. Эта информация обычно не меняется после того, как подключение установлено.

`PQsslInUse`

Возвращает true (1), если для текущего подключения используется SSL, и false (0) в противном случае.

```
int PQsslInUse(const PGconn *conn);
```

`PQsslAttribute`

Возвращает связанную с SSL информацию о соединении.

```
const char *PQsslAttribute(const PGconn *conn, const char *attribute_name);
```

Список доступных атрибутов зависит от применяемой библиотеки SSL и типа подключения. Если атрибут недоступен, возвращается NULL.

Обычно доступны следующие атрибуты:

`library`

Имя используемой реализации SSL. (В настоящее время поддерживается только `"OpenSSL"`)

`protocol`

Применяемая версия SSL/TLS. Наиболее распространены варианты `"TLSv1"`, `"TLSv1.1"` и `"TLSv1.2"`, но реализация может возвращать и другие строки, если применяется какой-то другой протокол.

`key_bits`

Число бит в ключе, используемом алгоритмом шифрования.

`cipher`

Краткое имя применяемого комплекта шифров, например: `"DHE-RSA-DES-CBC3-SHA"`. Эти имена могут быть разными в разных реализациях SSL.

`compression`

Возвращает `on`, если используется сжатие SSL, или `off` в противном случае.

`PQsslAttributeNames`

Возвращает массив имён доступных атрибутов SSL. Завершается массив указателем NULL.

```
const char * const * PQsslAttributeNames(const PGconn *conn);
```

`PQsslStruct`

Возвращает указатель на специфичный для реализации SSL объект, описывающий подключение.

```
void *PQsslStruct(const PGconn *conn, const char *struct_name);
```

Доступная структура (или структуры) зависит от того, какая реализация SSL применяется. Для OpenSSL выдаётся одна структура под именем «OpenSSL», и эта структура, на которую указывает возвращённый указатель, имеет тип `SSL` (определённый в OpenSSL). Для обращения к этой функции можно воспользоваться кодом в следующих строках:

```
#include <libpq-fe.h>
#include <openssl/ssl.h>

...

    SSL *ssl;

    dbconn = PQconnectdb(...);
    ...

    ssl = PQsslStruct(dbconn, "OpenSSL");
    if (ssl)
    {
        /* use OpenSSL functions to access ssl */
    }
```

Эта структура может использоваться, чтобы сличить уровни шифрования, проверить сертификаты сервера и т. д. За информацией об этой структуре обратитесь к документации по OpenSSL.

`PQgetssl`

Возвращает структуру SSL, использовавшуюся в соединении, или null, если SSL не используется.

```
void *PQgetssl(const PGconn *conn);
```

Эта функция равнозначна `PQsslStruct(conn, "OpenSSL")`. Её не следует применять в новых приложениях, так как возвращаемая структура специфична для OpenSSL и её нельзя будет получить с другой реализацией SSL. Чтобы проверить, использует ли подключение SSL, лучше вызвать `PQsslInUse`, а чтобы получить свойства подключения — `PQsslAttribute`.

31.3. Функции для исполнения команд

После того как соединение с сервером было успешно установлено, функции, описанные в этом разделе, используются для выполнения SQL-запросов и команд.

31.3.1. Главные функции

`PQexec`

Передаёт команду серверу и ожидает результата.

```
PGresult *PQexec(PGconn *conn, const char *command);
```

Возвращает указатель на `PGresult` или, возможно, пустой указатель (null). Как правило, возвращается непустой указатель, исключением являются ситуации нехватки памяти или серьёзные ошибки, такие, как невозможность отправки команды серверу. Для проверки возвращаемого значения на наличие ошибок следует вызывать функцию `PQresultStatus` (в случае нулевого указателя она возвратит `PGRES_FATAL_ERROR`). Для получения дополнительной информации о таких ошибках используйте функцию `PQerrorMessage`.

Строка команды может включать в себя более одной SQL-команды (которые разделяются точкой с запятой). Несколько запросов, отправленных с помощью одного вызова `PQexec`, обрабатываются в рамках одной транзакции, если только команды `BEGIN`/`COMMIT` не включены явно в строку запроса, чтобы разделить его на несколько транзакций. Однако обратите внимание, что возвращаемая структура `PGresult` описывает только результат последней из выполненных команд, содержащихся в строке запроса. Если одна из команд завершается сбоем, то обработка строки запроса на этом останавливается, и возвращённая структура `PGresult` описывает состояние ошибки.

`PQexecParams`

Отправляет команду серверу и ожидает результата. Имеет возможность передать параметры отдельно от текста SQL-команды.

```
PGresult *PQexecParams(PGconn *conn,
                       const char *command,
                       int nParams,
                       const Oid *paramTypes,
                       const char * const *paramValues,
                       const int *paramLengths,
                       const int *paramFormats,
                       int resultFormat);
```

`PQexecParams` подобна `PQexec`, но предлагает дополнительную функциональность: значения параметров могут быть указаны отдельно от самой строки-команды, а результаты запроса могут быть затребованы либо в текстовом, либо в двоичном формате. `PQexecParams` поддерживается только при подключениях по протоколу версии 3.0 или более поздних версий. Её вызов завершится сбоем при использовании протокола версии 2.0.

Параметры функции следующие:

conn

Объект, описывающий подключение, через которое пересылается команда.

command

Строка SQL-команды, которая должна быть выполнена. Если используются параметры, то в строке команды на них ссылаются, как `$1`, `$2` и т. д.

nParams

Число предоставляемых параметров. Оно равно длине массивов *paramTypes[]*, *paramValues[]*, *paramLengths[]* и *paramFormats[]*. (Указатели на массивы могут быть равны `NULL`, когда *nParams* равно нулю.)

paramTypes[]

Предписывает, посредством OID, типы данных, которые должны быть назначены параметрам. Если значение *paramTypes* равно `NULL` или какой-либо отдельный элемент в массиве равен нулю, тогда сервер самостоятельно определит тип данных для параметра точно таким же образом, как он сделал бы для литеральной строки, тип которой не указан.

paramValues[]

Указывает фактические значения параметров. Нулевой указатель в этом массиве означает, что соответствующий параметр равен null; в противном случае указатель указывает на текстовую строку, завершающуюся нулевым символом (для текстового формата), или на двоичные данные в формате, которого ожидает сервер (для двоичного формата).

paramLengths[]

Указывает фактические длины данных для параметров, представленных в двоичном формате. Он игнорируется для параметров, имеющих значение null, и для параметров, представленных в текстовом формате. Указатель на массив может быть нулевым, когда нет двоичных параметров.

paramFormats[]

Указывает, являются ли параметры текстовыми (поместите нуль в элемент массива, соответствующий такому параметру) или двоичными (поместите единицу в элемент массива, соответствующий такому параметру). Если указатель на массив является нулевым, тогда все параметры считаются текстовыми строками.

Значения, переданные в двоичном формате, требуют знания внутреннего представления, которого ожидает сервер. Например, целые числа должны передаваться с использованием сетевого порядка байтов. Передача значений типа `numeric` требует знания формата, в котором их хранит сервер; это реализовано в `src/backend/utils/adt/numeric.c::numeric_send()` и `src/backend/utils/adt/numeric.c::numeric_recv()`.

resultFormat

Требует указать ноль, чтобы получить результаты в текстовом формате, или единицу, чтобы получить результаты в двоичном формате. (В настоящее время нет возможности получить различные столбцы результата в разных форматах, хотя это и возможно на уровне протокола, лежащего в основе подключений.)

Важнейшим преимуществом `PQexecParams` над `PQexec` является то, что значения параметров могут быть отделены от строки-команды. Это позволяет избежать использования кавычек и экранирующих символов, что является трудоёмким методом, часто приводящим к ошибкам.

В отличие от `PQexec`, `PQexecParams` позволяет включать не более одной SQL-команды в строку запроса. (В ней могут содержаться точки с запятой, однако может присутствовать не более одной непустой команды.) Это ограничение накладывается базовым протоколом, но оно приносит и некоторую пользу в качестве дополнительной защиты от атак методом SQL-инъекций.

Подсказка

Указание типов параметров с помощью OID является трудоёмким, особенно если вы предпочитаете не указывать явно значений OID в вашей программе. Однако вы можете избежать этого даже в случаях, когда сервер самостоятельно не может определить тип параметра или выбирает не тот тип, который вы хотите. В строке SQL-команды добавьте явное приведение типа для этого параметра, чтобы показать, какой тип данных вы будете отправлять. Например:

```
SELECT * FROM mytable WHERE x = $1::bigint;
```

Это приведёт к тому, что параметр `$1` будет считаться имеющим тип `bigint`, в то время как по умолчанию ему был бы назначен тот же самый тип, что и `x`. Такое явное принятие решения о типе параметра либо с помощью описанного метода, либо путём задания числового OID строго рекомендуется, когда значения параметров отправляются в двоичном формате, поскольку двоичный формат имеет меньшую избыточность, чем текстовый, и поэтому гораздо менее вероятно, что сервер обнаружит ошибку несоответствия типов, допущенную вами.

`PQprepare`

Отправляет запрос, чтобы создать подготовленный оператор с конкретными параметрами, и ожидает завершения.

```
PGresult *PQprepare(PGconn *conn,
                    const char *stmtName,
                    const char *query,
                    int nParams,
                    const Oid *paramTypes);
```

`PQprepare` создаёт подготовленный оператор для последующего исполнения с помощью `PQexecPrepared`. Благодаря этому, команды, которые будут выполняться многократно, не потребуется разбирать и планировать каждый раз; за подробностями обратитесь к PREPARE. `PQprepare` поддерживается только с подключениями по протоколу 3.0 и новее; с протоколом 2.0 эта функция работать не будет.

Функция создаёт подготовленный оператор с именем *`stmtName`* из строки *`query`*, которая должна содержать единственную SQL-команду. *`stmtName`* может быть пустой строкой "", тогда будет создан неименованный оператор (в таком случае любой уже существующий неименованный оператор будет автоматически заменён), в противном случае, если имя оператора уже определено в текущем сеансе работы, будет ошибка. Если используются параметры, то в запросе к ним обращаются таким образом: `$1`, `$2` и т. д. *`nParams`* представляет число параметров, типы данных для которых указаны в массиве *`paramTypes[]`*. (Указатель на массив может быть равен `NULL`, когда значение *`nParams`* равно нулю.) *`paramTypes[]`* указывает, посредством OID, типы данных, которые будут назначены параметрам. Если *`paramTypes`* равен `NULL` или какой-либо элемент в этом массиве равен нулю, то сервер назначает тип данных соответствующему параметру точно таким же способом, как он сделал бы для литеральной строки, не имеющей типа. Также в запросе можно использовать параметры с номерами, большими, чем *`nParams`*; типы данных для них сервер также сможет подобрать. (См. описание `PQdescribePrepared`, где сказано, какие существуют средства, чтобы определить, какие типы данных были подобраны).

Как и при вызове `PQexec`, результатом является объект `PGresult`, содержимое которого показывает успех или сбой на стороне сервера. Нулевой указатель означает нехватку памяти или невозможность вообще отправить команду. Для получения дополнительной информации о таких ошибках используйте `PQerrorMessage`.

Подготовленные операторы для использования с `PQexecPrepared` можно также создать путём исполнения SQL-команд PREPARE. Также, хотя никакой функции libpq для удаления

подготовленного оператора не предусмотрено, для этой цели можно воспользоваться SQL-командой DEALLOCATE.

`PQexecPrepared`

Отправляет запрос на исполнение подготовленного оператора с данными параметрами и ожидает результата.

```
PGresult *PQexecPrepared(PGconn *conn,
                         const char *stmtName,
                         int nParams,
                         const char * const *paramValues,
                         const int *paramLengths,
                         const int *paramFormats,
                         int resultFormat);
```

`PQexecPrepared` подобна `PQexecParams`, но команда, подлежащая исполнению, указывается путём передачи имени предварительно подготовленного оператора вместо передачи строки запроса. Эта возможность позволяет командам, которые вызываются многократно, подвергаться разбору и планированию только один раз, а не при каждом их исполнении. Оператор должен быть подготовлен предварительно в рамках текущего сеанса работы. `PQexecPrepared` поддерживается только в соединениях по протоколу версии 3.0 или более поздних версий. При использовании протокола версии 2.0 функция завершится сбоем.

Параметры идентичны `PQexecParams`, за исключением того, что вместо строки запроса передаётся имя подготовленного оператора, и отсутствует параметр *`paramTypes[]`* (он не нужен, поскольку типы данных для параметров подготовленного оператора были определены при его создании).

`PQdescribePrepared`

Передаёт запрос на получение информации об указанном подготовленном операторе и ожидает завершения.

```
PGresult *PQdescribePrepared(PGconn *conn, const char *stmtName);
```

`PQdescribePrepared` позволяет приложению получить информацию о предварительно подготовленном операторе. `PQdescribePrepared` поддерживается только в соединениях по протоколу версии 3.0 или более поздних версий. При использовании протокола версии 2.0 функция завершится сбоем.

Для ссылки на неименованный оператор значение *`stmtName`* может быть пустой строкой `""` или `NULL`, в противном случае оно должно быть именем существующего подготовленного оператора. В случае успешного выполнения возвращается `PGresult` со статусом `PGRES_COMMAND_OK`. Функции `PQnparams` и `PQparamtype` позволяют извлечь из `PGresult` информацию о параметрах подготовленного оператора, а функции `PQnfields`, `PQfname`, `PQftype` и т. п. предоставляют информацию о результирующих столбцах (если они есть) данного оператора.

`PQdescribePortal`

Передаёт запрос на получение информации об указанном портале и ожидает завершения.

```
PGresult *PQdescribePortal(PGconn *conn, const char *portalName);
```

`PQdescribePortal` позволяет приложению получить информацию о предварительно созданном портале. (libpq не предоставляет прямого доступа к порталам, но вы можете использовать эту функцию для ознакомления со свойствами курсора, созданного с помощью SQL-команды `DECLARE CURSOR`.) `PQdescribePortal` поддерживается только в соединениях по протоколу версии 3.0 или более поздних версий. При использовании протокола версии 2.0 функция завершится сбоем.

Для ссылки на неименованный портал значение *`portalName`* может быть пустой строкой `""` или `NULL`, в противном случае оно должно быть именем существующего портала. В случае

успешного завершения возвращается PGresult со статусом PGRES_COMMAND_OK. С помощью функций PQnfields, PQfname, PQftype и т. д. можно извлечь из PGresult информацию о результирующих столбцах (если они есть) данного портала.

Структура PGresult содержит результат, возвращённый сервером. Разработчики приложений libpq должны тщательно поддерживать абстракцию PGresult. Для получения доступа к содержимому PGresult используйте функции доступа, описанные ниже. Избегайте непосредственного обращения к полям структуры PGresult, поскольку они могут измениться в будущем.

PQresultStatus

Возвращает статус результата выполнения команды.

```
ExecStatusType PQresultStatus(const PGresult *res);
```

PQresultStatus может возвращать одно из следующих значений:

PGRES_EMPTY_QUERY

Строка, отправленная серверу, была пустой.

PGRES_COMMAND_OK

Успешное завершение команды, не возвращающей никаких данных.

PGRES_TUPLES_OK

Успешное завершение команды, возвращающей данные (такой, как SELECT или SHOW).

PGRES_COPY_OUT

Начат перенос данных Copy Out (с сервера).

PGRES_COPY_IN

Начат перенос данных Copy In (на сервер).

PGRES_BAD_RESPONSE

Ответ сервера не был распознан.

PGRES_NONFATAL_ERROR

Произошла не фатальная ошибка (уведомление или предупреждение).

PGRES_FATAL_ERROR

Произошла фатальная ошибка.

PGRES_COPY_BOTH

Начат перенос данных Copy In/Out (на сервер и с сервера). Эта функция в настоящее время используется только для потоковой репликации, поэтому такой статус не должен иметь место в обычных приложениях.

PGRES_SINGLE_TUPLE

Структура PGresult содержит только одну результирующую строку, возвращённую текущей командой. Этот статус имеет место только тогда, когда для данного запроса был выбран режим построчного вывода (см. Раздел 31.5).

Если статус результата PGRES_TUPLES_OK или PGRES_SINGLE_TUPLE, тогда для извлечения строк, возвращённых запросом, можно использовать функции, описанные ниже. Обратите внимание, что команда SELECT, даже когда она не извлекает ни одной строки, всё же показывает PGRES_TUPLES_OK. PGRES_COMMAND_OK предназначен для команд, которые никогда не возвращают строки (INSERT или UPDATE без использования предложения RETURNING и др.).

Ответ PGRES_EMPTY_QUERY может указывать на наличие ошибки в клиентском программном обеспечении.

Результат со статусом PGRES_NONFATAL_ERROR никогда не будет возвращён напрямую функцией PQexec или другими функциями исполнения запросов; вместо этого результаты такого вида передаются обработчику уведомлений (см. Раздел 31.12).

PQresStatus

Преобразует значение перечислимого типа, возвращённое функцией PQresultStatus, в строковую константу, описывающую код статуса. Вызывающая функция не должна освобождать память, на которую указывает возвращаемый указатель.

```
char *PQresStatus(ExecStatusType status);
```

PQresultErrorMessage

Возвращает сообщение об ошибке, связанное с командой, или пустую строку, если ошибки не произошло.

```
char *PQresultErrorMessage(const PGresult *res);
```

Если произошла ошибка, то возвращённая строка будет включать завершающий символ новой строки. Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель. Она будет освобождена, когда соответствующий указатель PGresult будет передан функции PQclear.

Если непосредственно после вызова PQexec или PQgetResult вызвать функцию PQerrorMessage (для данного подключения), то она возвратит ту же самую строку, что и PQresultErrorMessage (для данного результата). Однако PGresult сохранит своё сообщение об ошибке до тех пор, пока не будет уничтожен, в то время, как сообщение об ошибке, связанное с данным подключением, будет изменяться при выполнении последующих операций. Воспользуйтесь функцией PQresultErrorMessage, когда вы хотите узнать статус, связанный с конкретной структурой PGresult; используйте функцию PQerrorMessage, когда вы хотите узнать статус выполнения самой последней операции на данном соединении.

PQresultVerboseErrorMessage

Возвращает переформатированную версию сообщения об ошибке, связанного с объектом PGresult.

```
char *PQresultVerboseErrorMessage(const PGresult *res,
                                  PGVerbosity verbosity,
                                  PGContextVisibility show_context);
```

В некоторых ситуациях клиент может захотеть получить более подробную версию ранее выданного сообщения об ошибке. Эту потребность удовлетворяет функция PQresultVerboseErrorMessage, формируя сообщение, которое было бы выдано функцией PQresultErrorMessage, если бы заданный уровень детализации был текущим для соединения в момент заполнения PGresult. Если же в PGresult не содержится ошибка, вместо этого выдаётся сообщение «PGresult is not an error result» (PGresult — не результат с ошибкой). Возвращаемое этой функцией сообщение завершается переводом строки.

В отличие от многих других функций, извлекающих данные из PGresult, результат этой функции — новая размещённая в памяти строка. Когда эта строка будет не нужна, вызывающий код должен освободить её место, вызвав PQfreemem().

При нехватке памяти может быть возвращёно NULL.

PQresultErrorField

Возвращает индивидуальное поле из отчёта об ошибке.

```
char *PQresultErrorField(const PGresult *res, int fieldcode);
```

fieldcode это идентификатор поля ошибки; см. символические константы, перечисленные ниже. Если PGresult не содержит ошибки или предупреждения или не включает указанное поле, то возвращается NULL. Значения полей обычно не включают завершающий символ новой строки. Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель. Она будет освобождена, когда соответствующий указатель PGresult будет передан функции PQclear.

Доступны следующие коды полей:

PG_DIAG_SEVERITY

Серьёзность; поле может содержать ERROR, FATAL или PANIC (в сообщении об ошибке) либо WARNING, NOTICE, DEBUG, INFO или LOG (в сообщении-уведомлении) либо локализованный перевод одного из этих значений. Присутствует всегда.

PG_DIAG_SEVERITY_NONLOCALIZED

Серьёзность; поле может содержать ERROR, FATAL или PANIC (в сообщении об ошибке) либо WARNING, NOTICE, DEBUG, INFO или LOG (в сообщении-уведомлении). Это поле подобно PG_DIAG_SEVERITY, но его содержимое никогда не переводится. Присутствует только в отчётах, выдаваемых Postgres Pro версии 9.6 и новее.

PG_DIAG_SQLSTATE

Код ошибки в соответствии с соглашением о кодах SQLSTATE. Код SQLSTATE идентифицирует тип случившейся ошибки; он может использоваться клиентскими приложениями, чтобы выполнять конкретные операции (такие, как обработка ошибок) в ответ на конкретную ошибку базы данных. Список возможных кодов SQLSTATE приведён в Приложении A. Это поле не подлежит локализации. Оно всегда присутствует.

PG_DIAG_MESSAGE_PRIMARY

Главное сообщение об ошибке, предназначенное для прочтения пользователем. Как правило составляет всего одну строку. Это поле всегда присутствует.

PG_DIAG_MESSAGE_DETAIL

Необязательное дополнительное сообщение об ошибке, передающее более детальную информацию о проблеме. Может занимать несколько строк.

PG_DIAG_MESSAGE_HINT

Подсказка: необязательное предположение о том, что можно сделать в данной проблемной ситуации. Оно должно отличаться от детальной информации в том смысле, что оно предлагает совет (возможно, и неподходящий), а не просто факты. Может занимать несколько строк.

PG_DIAG_STATEMENT_POSITION

Строка, содержащая десятичное целое число, указывающее позицию расположения ошибки в качестве индекса в оригинальной строке оператора. Первый символ имеет позицию 1, при этом позиции измеряются в символах а не в байтах.

PG_DIAG_INTERNAL_POSITION

Это поле определяется точно так же, как и поле PG_DIAG_STATEMENT_POSITION, но оно используется, когда позиция местонахождения ошибки относится к команде, сгенерированной внутренними модулями, а не к команде, представленной клиентом. Когда появляется это поле, то всегда появляется и поле PG_DIAG_INTERNAL_QUERY.

PG_DIAG_INTERNAL_QUERY

Текст команды, сгенерированной внутренними модулями, завершившейся сбоем. Это мог бы быть, например, SQL-запрос, выданный функцией на языке PL/pgSQL.

PG_DIAG_CONTEXT

Характеристика контекста, в котором произошла ошибка. В настоящее время она включает вывод стека вызовов активных функций процедурного языка и запросов, сгенерированных внутренними модулями. Стек выводится по одному элементу в строке, при этом первым идет самый последний из элементов (самый недавний вызов).

PG_DIAG_SCHEMA_NAME

Если ошибка была связана с конкретным объектом базы данных, то в это поле будет записано имя схемы, содержащей данный объект.

PG_DIAG_TABLE_NAME

Если ошибка была связана с конкретной таблицей, то в это поле будет записано имя таблицы. (Для получения имени схемы для данной таблицы обратитесь к полю, содержащему имя схемы.)

PG_DIAG_COLUMN_NAME

Если ошибка была связана с конкретным столбцом таблицы, то в это поле будет записано имя столбца. (Чтобы идентифицировать таблицу, обратитесь к полям, содержащим имена схемы и таблицы.)

PG_DIAG_DATATYPE_NAME

Если ошибка была связана с конкретным типом данных, то в это поле будет записано имя типа данных. (Чтобы получить имя схемы, которой принадлежит этот тип данных, обратитесь к полю, содержащему имя схемы.)

PG_DIAG_CONSTRAINT_NAME

Если ошибка была связана с конкретным ограничением, то в это поле будет записано имя ограничения. Чтобы получить имя таблицы или домена, связанных с этим ограничением, обратитесь к полям, перечисленным выше. (С этой целью индексы рассматриваются как ограничения, даже если они и не были созданы с помощью синтаксиса для создания ограничений.)

PG_DIAG_SOURCE_FILE

Имя файла, содержащего позицию в исходном коде, для которой было выдано сообщение об ошибка.

PG_DIAG_SOURCE_LINE

Номер строки той позиции в исходном коде, для которой было выдано сообщение об ошибке.

PG_DIAG_SOURCE_FUNCTION

Имя функции в исходном коде, сообщающей об ошибке.

Примечание

Поля для имени схемы, имени таблицы, имени столбца, имени типа данных и имени ограничения предоставляются лишь для ограниченного числа типов ошибок; см. Приложение A. Не рассчитывайте на то, что присутствие любого из этих полей гарантирует и присутствие какого-то другого поля. Базовые источники ошибок придерживаются взаимосвязей, описанных выше, но функции, определённые пользователем, могут использовать эти поля другими способами. Аналогично, не рассчитывайте на то, что эти поля обозначают объекты, существующие в текущей базе данных в настоящий момент.

Клиент отвечает за форматирование отображаемой информации в соответствии с его нуждами; в частности, он должен разбивать длинные строки, как требуется. Символы новой строки, встречающиеся в полях сообщения об ошибке, должны обрабатываться, как разрывы абзацев, а не строк.

Ошибки, сгенерированные внутренними модулями libpq, будут иметь поля серьёзности ошибки и основного сообщения, но, как правило, никаких других полей. Ошибки, возвращаемые сервером, работающим по протоколу версии ниже 3.0, будут включать поля серьёзности ошибки и основного сообщения, а также иногда детальное сообщение, но больше никаких полей.

Заметьте, что поля ошибки доступны только из объектов `PGresult`, а не из объектов `PGconn`. Не существует функции `PQerrorField`.

`PQclear`

Освобождает область памяти, связанную с `PGresult`. Результат выполнения каждой команды должен быть освобождён с помощью `PQclear`, когда он больше не нужен.

```
void PQclear(PGresult *res);
```

Вы можете держать объект `PGresult` под рукой до тех пор, пока он вам нужен; он не исчезает, ни когда вы выдаёте новую команду, ни даже если вы закрываете соединение. Чтобы от него избавиться, вы должны вызвать `PQclear`. Если этого не делать, то в результате будут иметь место утечки памяти в вашем приложении.

31.3.2. Извлечение информации, связанной с результатом запроса

Эти функции служат для извлечения информации из объекта `PGresult`, который представляет результат успешного запроса (то есть такого, который имеет статус `PGRES_TUPLES_OK` или `PGRES_SINGLE_TUPLE`). Их также можно использовать для извлечения информации об успешной операции DESCRIBE: результат этой операции содержит всю ту же самую информацию о столбцах, которая была бы получена при реальном исполнении запроса, но не содержит ни одной строки. Для объектов, имеющих другие значения статуса, эти функции будут действовать таким образом, как будто результат не содержит ни одной строки и ни одного столбца.

`PQntuples`

Возвращает число строк (кортежей) в полученной выборке. (Заметьте, что объекты `PGresult` не могут содержать более чем `INT_MAX` строк, так что для результата достаточно типа `int`.)

```
int PQntuples(const PGresult *res);
```

`PQnfields`

Возвращает число столбцов (полей) в каждой строке полученной выборки.

```
int PQnfields(const PGresult *res);
```

`PQfname`

Возвращает имя столбца, соответствующего данному номеру столбца. Номера столбцов начинаются с 0. Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель. Она будет освобождена, когда соответствующий указатель на `PGresult` будет передан функции `PQclear`.

```
char *PQfname(const PGresult *res,
              int column_number);
```

Если номер столбца выходит за пределы допустимого диапазона, то возвращается `NULL`.

`PQfnumber`

Возвращает номер столбца, соответствующий данному имени столбца.

```
int PQfnumber(const PGresult *res,
              const char *column_name);
```

Если данное имя не совпадает с именем ни одного из столбцов, то возвращается -1.

Данное имя интерпретируется, как идентификатор в SQL-команде. Это означает, что оно переводится в нижний регистр, если только оно не заключено в двойные кавычки. Например, для выборки, сгенерированной с помощью такой SQL-команды:

```
SELECT 1 AS FOO, 2 AS "BAR";
```

мы получили бы следующие результаты:

```
PQfname(res, 0)              foo
PQfname(res, 1)              BAR
PQfnumber(res, "FOO")        0
PQfnumber(res, "foo")        0
PQfnumber(res, "BAR")        -1
PQfnumber(res, "\"BAR\"")    1
```

`PQftable`

Возвращает OID таблицы, из которой был получен данный столбец. Номера столбцов начинаются с 0.

```
Oid PQftable(const PGresult *res,
             int column_number);
```

В следующих случаях возвращается `InvalidOid`: если номер столбца выходит за пределы допустимого диапазона; если указанный столбец не является простой ссылкой на столбец таблицы; когда используется протокол версии более ранней, чем 3.0. Вы можете сделать запрос к системной таблице `pg_class`, чтобы точно определить, к какой таблице было произведено обращение.

Тип данных `Oid` и константа `InvalidOid` будут определены, когда вы включите заголовочный файл для libpq. Они будут принадлежать к одному из целочисленных типов.

`PQftablecol`

Возвращает номер столбца (в пределах его таблицы) для указанного столбца в полученной выборке. Номера столбцов в полученной выборке начинаются с 0, но столбцы в таблице имеют ненулевые номера.

```
int PQftablecol(const PGresult *res,
                int column_number);
```

В следующих случаях возвращается ноль: если номер столбца выходит за пределы допустимого диапазона; если указанный столбец не является простой ссылкой на столбец таблицы; когда используется протокол версии более ранней, чем 3.0.

`PQfformat`

Возвращает код формата, показывающий формат данного столбца. Номера столбцов начинаются с 0.

```
int PQfformat(const PGresult *res,
              int column_number);
```

Значение кода формата, равное нулю, указывает на текстовое представление данных, в то время, как значение, равное единице, означает двоичное представление. (Другие значения кодов зарезервированы для определения в будущем.)

`PQftype`

Возвращает тип данных, соответствующий данному номеру столбца. Возвращаемое целое значение является внутренним номером OID для этого типа. Номера столбцов начинаются с 0.

```
Oid PQftype(const PGresult *res,
            int column_number);
```

Вы можете сделать запрос к системной таблице `pg_type`, чтобы получить имена и свойства различных типов данных. Значения OID для встроенных типов данных определены в файле `include/server/catalog/pg_type.h` в каталоге установленного сервера.

`PQfmod`

Возвращает модификатор типа для столбца, соответствующего данному номеру. Номера столбцов начинаются с 0.

```
int PQfmod(const PGresult *res,
           int column_number);
```

Интерпретация значений модификатора зависит от типа; они обычно показывают точность или предельные размеры. Значение -1 используется, чтобы показать «нет доступной информации». Большинство типов данных не используют модификаторов, в таком случае значение всегда будет -1.

`PQfsize`

Возвращает размер в байтах для столбца, соответствующего данному номеру. Номера столбцов начинаются с 0.

```
int PQfsize(const PGresult *res,
            int column_number);
```

`PQfsize` возвращает размер пространства, выделенного для этого столбца в строке базы данных, другими словами, это размер внутреннего представления этого типа данных на сервере. (Следовательно, эта информация не является по-настоящему полезной для клиентов.) Отрицательное значение говорит о том, что тип данных имеет переменную длину.

`PQbinaryTuples`

Возвращает 1, если `PGresult` содержит двоичные данные, или 0, если данные текстовые.

```
int PQbinaryTuples(const PGresult *res);
```

Эта функция не рекомендуется к использованию (за исключением применения в связи с командой `COPY`), поскольку один и тот же `PGresult` может содержать в некоторых столбцах текстовые данные, а в остальных — двоичные. Предпочтительнее использовать `PQfformat`. `PQbinaryTuples` возвращает 1, только если все столбцы в выборке являются двоичными (код формата 1).

`PQgetvalue`

Возвращает значение одного поля из одной строки, содержащейся в `PGresult`. Номера строк и столбцов начинаются с 0. Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель. Она будет освобождена, когда соответствующий указатель на `PGresult` будет передан функции `PQclear`.

```
char *PQgetvalue(const PGresult *res,
                 int row_number,
                 int column_number);
```

Для данных в текстовом формате значение, возвращаемое функцией `PQgetvalue`, является значением поля, представленным в виде символьной строки с завершающим нулевым символом. Для данных в двоичном формате используется двоичное представление значения. Оно определяется функциями `typsend` и `typreceive` для конкретного типа данных. (В этом случае к значению также добавляется нулевой байт, но обычно это не приносит пользы, поскольку вероятно, что значение уже содержит нулевые байты.)

Пустая строка возвращается в том случае, когда значение поля отсутствует (null). См. PQgetisnull, чтобы отличать отсутствие значения (null) от значения, равного пустой строке.

Указатель, возвращаемый функцией PQgetvalue, указывает на область хранения, которая является частью структуры PGresult. Не следует модифицировать данные, на которые указывает этот указатель, а нужно явно скопировать данные в другую область хранения, если предполагается их использовать за пределами времени жизни самой структуры PGresult.

PQgetisnull

Проверяет поле на предмет отсутствия значения (null). Номера строк и столбцов начинаются с 0.

```
int PQgetisnull(const PGresult *res,
                int row_number,
                int column_number);
```

Эта функция возвращает 1, если значение в поле отсутствует (null), и 0, если поле содержит непустое (non-null) значение. (Обратите внимание, что PQgetvalue возвратит пустую строку, а не нулевой указатель, если значение в поле отсутствует.)

PQgetlength

Возвращает фактическую длину значения поля в байтах. Номера строк и столбцов начинаются с 0.

```
int PQgetlength(const PGresult *res,
                int row_number,
                int column_number);
```

Это фактическая длина данных для конкретного значения данных, то есть размер объекта, на который указывает PQgetvalue. Для текстового формата данных это то же самое, что strlen(). Для двоичного же формата это существенная информация. Обратите внимание, что *не* следует полагаться на PQfsize, чтобы получить фактическую длину данных.

PQnparams

Возвращает число параметров подготовленного оператора.

```
int PQnparams(const PGresult *res);
```

Эта функция полезна только при исследовании результата работы функции PQdescribePrepared. Для других типов запросов она возвратит ноль.

PQparamtype

Возвращает тип данных для указанного параметра оператора. Номера параметров начинаются с 0.

```
Oid PQparamtype(const PGresult *res, int param_number);
```

Эта функция полезна только при исследовании результата работы функции PQdescribePrepared. Для других типов запросов она возвратит ноль.

PQprint

Выводит все строки и, по выбору, имена столбцов в указанный поток вывода.

```
void PQprint(FILE *fout,      /* поток вывода */
             const PGresult *res,
             const PQprintOpt *po);
typedef struct
{
    pqbool  header;      /* печатать заголовки полей и счётчик строк */
    pqbool  align;       /* выравнивать поля */
```

```
    pqbool  standard;    /* старый формат */
    pqbool  html3;       /* выводить HTML-таблицы */
    pqbool  expanded;    /* расширять таблицы */
    pqbool  pager;       /* использовать программу для постраничного просмотра, если
 нужно */
    char    *fieldSep;   /* разделитель полей */
    char    *tableOpt;   /* атрибуты для HTML-таблицы */
    char    *caption;    /* заголовок HTML-таблицы */
    char    **fieldName; /* массив заменителей для имён полей, завершающийся нулевым
 символом */
} PQprintOpt;
```

Эту функцию прежде использовала утилита psql для вывода результатов запроса, но больше она её не использует. Обратите внимание, предполагается, что все данные представлены в текстовом формате.

31.3.3. Получение другой информации о результате

Эти функции используются для получения остальной информации из объектов `PGresult`.

`PQcmdStatus`

Возвращает дескриптор статуса для SQL-команды, которая сгенерировала `PGresult`.

```
char *PQcmdStatus(PGresult *res);
```

Как правило, это просто имя команды, но могут быть включены и дополнительные сведения, такие, как число обработанных строк. Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель. Она будет освобождена, когда соответствующий указатель на `PGresult` будет передан функции `PQclear`.

`PQcmdTuples`

Возвращает число строк, которые затронула SQL-команда.

```
char *PQcmdTuples(PGresult *res);
```

Эта функция возвращает строковое значение, содержащее число строк, которые затронул SQL-оператор, сгенерировавший данный `PGresult`. Эту функцию можно использовать только сразу после выполнения команд `SELECT`, `CREATE TABLE AS`, `INSERT`, `UPDATE`, `DELETE`, `MOVE`, `FETCH` или `COPY`, а также после оператора `EXECUTE`, выполнившего подготовленный запрос, содержащий команды `INSERT`, `UPDATE` или `DELETE`. Если команда, которая сгенерировала `PGresult`, была какой-то иной, то `PQcmdTuples` возвращает пустую строку. Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель. Она будет освобождена, когда соответствующий указатель на `PGresult` будет передан функции `PQclear`.

`PQoidValue`

Возвращает OID вставленной строки, если SQL-команда была командой `INSERT`, которая вставила ровно одну строку в таблицу, имеющую идентификаторы OID, или командой `EXECUTE`, которая выполнила подготовленный запрос, содержащий соответствующий оператор `INSERT`. В противном случае эта функция возвращает `InvalidOid`. Эта функция также возвратит `InvalidOid`, если таблица, затронутая командой `INSERT`, не содержит идентификаторов OID.

```
Oid PQoidValue(const PGresult *res);
```

`PQoidStatus`

Эта функция считается не рекомендуемой к использованию (в качестве замены служит `PQoidValue`), а также она не отвечает требованиям потоковой безопасности. Она возвращает строковое значение, содержащее OID вставленной строки, в то время как `PQoidValue` возвращает значение OID.

```
char *PQoidStatus(const PGresult *res);
```

31.3.4. Экранирование строковых значений для включения в SQL-команды

`PQescapeLiteral`

```
char *PQescapeLiteral(PGconn *conn, const char *str, size_t length);
```

`PQescapeLiteral` экранирует строковое значение для использования внутри SQL-команды. Это полезно при вставке в SQL-команды значений данных в виде литеральных констант. Определённые символы (такие, как кавычки и символы обратной косой черты) должны экранироваться, чтобы предотвратить их специальную интерпретацию синтаксическим анализатором языка SQL. `PQescapeLiteral` выполняет эту операцию.

`PQescapeLiteral` возвращает экранированную версию параметра *`str`*, размещённую в области памяти, распределённой с помощью функции `malloc()`. Эту память нужно освобождать с помощью функции `PQfreemem()`, когда возвращённое значение больше не требуется. Завершающий нулевой байт не нужен и не должен учитываться в параметре *`length`*. (Если завершающий нулевой байт был найден до того, как были обработаны *`length`* байт, то `PQescapeLiteral` останавливает работу на нулевом байте; таким образом, поведение функции напоминает `strncpy`.) В возвращённой строке все специальные символы заменены таким образом, что синтаксический анализатор строковых литералов Postgres Pro может обработать их должным образом. Завершающий нулевой байт также будет добавлен. Одинарные кавычки, которые должны окружать строковые литералы Postgres Pro, включаются в результирующую строку.

В случае ошибки `PQescapeLiteral` возвращает `NULL`, и в объект *`conn`* помещается соответствующее сообщение.

> **Подсказка**
>
> Особенно важно выполнять надлежащее экранирование при обработке строк, полученных из ненадёжных источников. В противном случае ваша безопасность подвергается риску из-за уязвимости в отношении атак с использованием «SQL-инъекций», с помощью которых нежелательные SQL-команды направляются в вашу базу данных.

Обратите внимание, что нет необходимости (и это будет даже некорректно) экранировать значения данных, передаваемых в виде отдельных параметров в функцию `PQexecParams` или родственные ей функции.

`PQescapeIdentifier`

```
char *PQescapeIdentifier(PGconn *conn, const char *str, size_t length);
```

`PQescapeIdentifier` экранирует строку, предназначенную для использования в качестве идентификатора SQL, такого, как таблица, столбец или имя функции. Это полезно, когда идентификатор, выбранный пользователем, может содержать специальные символы, которые в противном случае не интерпретировались бы синтаксическим анализатором SQL, как часть идентификатора, или когда идентификатор может содержать символы верхнего регистра, и этот регистр требуется сохранить.

`PQescapeIdentifier` возвращает версию параметра *`str`*, экранированную как SQL-идентификатор, и размещённую в области памяти, распределённой с помощью функции `malloc()`. Эту память нужно освобождать с помощью функции `PQfreemem()`, когда возвращённое значение больше не требуется. Завершающий нулевой байт не нужен и не должен учитываться в параметре *`length`*. (Если завершающий нулевой байт был найден до

того, как были обработаны *length* байт, то PQescapeIdentifier останавливает работу на нулевом байте; таким образом, поведение функции напоминает strncpy.) В возвращённой строке все специальные символы заменены таким образом, что она будет надлежащим образом обработана, как SQL-идентификатор. Завершающий нулевой байт также будет добавлен. Возвращённая строка также будет заключена в двойные кавычки.

В случае ошибки PQescapeIdentifier возвращает NULL, и в объект *conn* помещается соответствующее сообщение.

> **Подсказка**
>
> Как и в случае со строковыми литералами, для того чтобы предотвратить атаки с помощью SQL-инъекций, SQL-идентификаторы должны экранироваться, когда они получены из ненадёжного источника.

PQescapeStringConn

```
size_t PQescapeStringConn(PGconn *conn,
                          char *to, const char *from, size_t length,
                          int *error);
```

PQescapeStringConn экранирует строковые литералы наподобие PQescapeLiteral. Но, в отличие от PQescapeLiteral, за предоставление буфера надлежащего размера отвечает вызывающая функция. Более того, PQescapeStringConn не добавляет одинарные кавычки, которые должны окружать строковые литералы Postgres Pro; они должны быть включены в SQL-команду, в которую вставляется результирующая строка. Параметр *from* указывает на первый символ строки, которая должна экранироваться, а параметр *length* задаёт число байт в этой строке. Завершающий нулевой байт не требуется и не должен учитываться в параметре *length*. (Если завершающий нулевой байт был найден до того, как были обработаны *length* байт, то PQescapeStringConn останавливает работу на нулевом байте; таким образом, поведение функции напоминает strncpy.) Параметр *to* должен указывать на буфер, который сможет вместить как минимум на один байт больше, чем предписывает удвоенное значение параметра *length*, в противном случае поведение функции не определено. Поведение будет также не определено, если строки *to* и *from* перекрываются.

Если параметр *error* не равен NULL, тогда значение *error устанавливается равным нулю в случае успешной работы и не равным нулю в случае ошибки. В настоящее время единственным возможным условием возникновения ошибки является неверная мультибайтовая кодировка в исходной строке. Выходная строка формируется даже при наличии ошибки, но можно ожидать, что сервер отвергнет её как неверно сформированную. В случае ошибки в объект *conn* записывается соответствующее сообщение независимо от того, равно ли NULL значение параметра *error*.

PQescapeStringConn возвращает число байт, записанных по адресу *to*, не включая завершающий нулевой байт.

PQescapeString

PQescapeString является более старой, не рекомендованной к использованию версией функции PQescapeStringConn.

```
size_t PQescapeString (char *to, const char *from, size_t length);
```

Единственное отличие от PQescapeStringConn состоит в том, что функция PQescapeString не принимает PGconn или *error* в качестве параметров. Из-за этого она не может скорректировать своё поведение в зависимости от свойств подключения (таких, как кодировка символов) и, следовательно, *она может выдавать неверные результаты*. Также она не имеет способа сообщить об ошибках.

PQescapeString может безопасно использоваться в клиентских программах, которые работают лишь с одним подключением к Postgres Pro за один раз (в этом случае функция может найти то, что ей нужно знать, «за кулисами»). В других контекстах её использование несёт в себе угрозу безопасности и его следует избегать в пользу применения функции PQescapeStringConn.

PQescapeByteaConn

Экранирует двоичные данные для их использования внутри SQL-команды с типом данных bytea. Как и в случае с PQescapeStringConn, эта функция применяется только тогда, когда данные вставляются непосредственно в строку SQL-команды.

```
unsigned char *PQescapeByteaConn(PGconn *conn,
                                 const unsigned char *from,
                                 size_t from_length,
                                 size_t *to_length);
```

Байты, имеющие определённые значения, должны экранироваться, когда они используются в качестве составной части литерала, имеющего тип bytea, в SQL-операторе. PQescapeByteaConn экранирует байты, используя либо hex-кодирование, либо экранирование с помощью обратной косой черты. См. Раздел 8.4 для получения дополнительной информации.

Параметр *from* указывает на первый байт строки, которая должна экранироваться, а параметр *from_length* задаёт число байт в этой двоичной строке. (Завершающий нулевой байт не нужен и не учитывается.) Параметр *to_length* указывает на переменную, которая будет содержать длину результирующей экранированной строки. Эта длина включает завершающий нулевой байт результирующей строки.

PQescapeByteaConn возвращает экранированную версию двоичной строки, на которую указывает параметр *from*, и размещает её в памяти, распределённой с помощью malloc(). Эта память должна быть освобождена с помощью функции PQfreemem(), когда результирующая строка больше не нужна. В возвращаемой строке все специальные символы заменены так, чтобы синтаксический анализатор литеральных строк Postgres Pro и функция ввода для типа bytea могли обработать их надлежащим образом. Завершающий нулевой байт также добавляется. Одинарные кавычки, которые должны окружать строковые литералы Postgres Pro, не являются частью результирующей строки.

В случае ошибки возвращается нулевой указатель, и соответствующее сообщение об ошибке записывается в объект *conn*. В настоящее время единственной возможной ошибкой может быть нехватка памяти для результирующей строки.

PQescapeBytea

PQescapeBytea является более старой, не рекомендуемой к использованию версией функции PQescapeByteaConn.

```
unsigned char *PQescapeBytea(const unsigned char *from,
                             size_t from_length,
                             size_t *to_length);
```

Единственное отличие от PQescapeByteaConn состоит в том, что функция PQescapeBytea не принимает PGconn в качестве параметра. Из-за этого PQescapeBytea может безопасно использоваться в клиентских программах, которые работают лишь с одним подключением к Postgres Pro за один раз (в этом случае функция может найти то, что ей нужно знать, «за кулисами»). Она *может выдавать неверные результаты* при использовании в программах, которые формируют множественные подключения к базе данных (в таких случаях используйте PQescapeByteaConn).

PQunescapeBytea

Преобразует строковое представление двоичных данных в двоичные данные — является обратной функцией к функции PQescapeBytea. Она нужна, когда данные типа bytea извлекаются в текстовом формате, но не когда они извлекаются в двоичном формате.

```
unsigned char *PQunescapeBytea(const unsigned char *from, size_t *to_length);
```

Параметр *from* указывает на строку, такую, какую могла бы возвратить функция PQgetvalue, применённая к столбцу типа bytea. PQunescapeBytea преобразует это строковое представление в его двоичное представление. Она возвращает указатель на буфер, распределённый с помощью функции malloc() (или NULL в случае ошибки) и помещает размер буфера по адресу *to_length*. Когда результат не будет нужен, необходимо освободить его память, вызвав PQfreemem.

Это преобразование не является точной инверсией для PQescapeBytea, поскольку ожидается, что строка, полученная от PQgetvalue, не будет «экранированной». В частности, это означает, что учитывать режим спецпоследовательностей не нужно, и поэтому в параметре нет необходимости PGconn.

31.4. Асинхронная обработка команд

Функция PQexec хорошо подходит для отправки команд серверу в нормальных, синхронных приложениях. Однако она имеет ряд недостатков, которые могут иметь значение для некоторых пользователей:

- PQexec ожидает завершения выполнения команды. Приложение может иметь и другую работу, которую нужно делать (такую, как поддержание пользовательского интерфейса), в таком случае оно не захочет блокироваться, ожидая ответа.
- Поскольку выполнение клиентского приложения приостанавливается, пока оно ожидает результата, то приложению трудно решить, что оно хотело бы попытаться отменить выполняющуюся команду. (Это можно сделать из обработчика сигнала, но никак иначе.)
- PQexec может возвратить только одну структуру PGresult. Если отправленная серверу строка-команда содержит множественные SQL-команды, то все структуры PGresult, кроме последней, отбрасываются функцией PQexec.
- PQexec всегда собирает все результаты выполнения команды, буферизуя их в единственной структуре PGresult. В то время как для приложения это упрощает логику обработки ошибок, это может быть непрактично, когда результат содержит много строк.

Приложения, которым эти ограничения не подходят, могут вместо PQexec использовать базовые функции, на основе которых и построена функция PQexec: PQsendQuery и PQgetResult. Есть также функции PQsendQueryParams, PQsendPrepare, PQsendQueryPrepared, PQsendDescribePrepared и PQsendDescribePortal, которые можно использовать совместно с PQgetResult, чтобы продублировать функциональность PQexecParams, PQprepare, PQexecPrepared, PQdescribePrepared и PQdescribePortal соответственно.

PQsendQuery

Отправляет команду серверу, не ожидая получения результата. Если команда была отправлена успешно, то функция возвратит значение 1, в противном случае она возвратит 0 (тогда нужно воспользоваться функцией PQerrorMessage для получения дополнительной информации о сбое).

```
int PQsendQuery(PGconn *conn, const char *command);
```

После успешного вызова PQsendQuery вызовите PQgetResult один или несколько раз, чтобы получить результат. Функцию PQsendQuery нельзя вызвать повторно (на том же самом соединении) до тех пор, пока PQgetResult не вернёт нулевой указатель, означающий, что выполнение команды завершено.

PQsendQueryParams

Отправляет серверу команду и обособленные параметры, не ожидая получения результатов.

```
int PQsendQueryParams(PGconn *conn,
```

```
                        const char *command,
                        int nParams,
                        const Oid *paramTypes,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```

Эта функция эквивалентна функции `PQsendQuery`, за исключением того, что параметры запроса можно указать отдельно от самой строки запроса. Эта функция обрабатывает свои параметры точно так же, как и функция `PQexecParams`. Аналогично функции `PQexecParams`, данная функция не будет работать при подключениях по протоколу версии 2.0, также она позволяет включить только одну команду в строку запроса.

`PQsendPrepare`

Посылает запрос на создание подготовленного оператора с данными параметрами и не дожидается завершения его выполнения.

```
int PQsendPrepare(PGconn *conn,
                  const char *stmtName,
                  const char *query,
                  int nParams,
                  const Oid *paramTypes);
```

Это асинхронная версия функции `PQprepare`. Она возвращает 1, если ей удалось отправить запрос, и 0 в противном случае. После её успешного вызова следует вызвать функцию `PQgetResult`, чтобы определить, успешно ли создал сервер подготовленный оператор. Эта функция обрабатывает свои параметры точно так же, как и функция `PQprepare`. Аналогично функции `PQprepare`, данная функция не будет работать при подключениях по протоколу версии 2.0.

`PQsendQueryPrepared`

Посылает запрос на выполнение подготовленного оператора с данными параметрами, не ожидая получения результата.

```
int PQsendQueryPrepared(PGconn *conn,
                        const char *stmtName,
                        int nParams,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```

Эта функция подобна функции `PQsendQueryParams`, но команда, которая должна быть выполнена, задаётся путём указания предварительно подготовленного оператора, вместо задания строки запроса. Эта функция обрабатывает свои параметры точно так же, как и функция `PQexecPrepared`. Аналогично функции `PQexecPrepared`, данная функция не будет работать при подключениях по протоколу версии 2.0.

`PQsendDescribePrepared`

Отправляет запрос на получение информации об указанном подготовленном операторе и не дожидается завершения выполнения запроса.

```
int PQsendDescribePrepared(PGconn *conn, const char *stmtName);
```

Это асинхронная версия функции `PQdescribePrepared`. Она возвращает 1, если ей удалось отправить запрос, и 0 в противном случае. После её успешного вызова следует вызвать функцию `PQgetResult` для получения результата. Эта функция обрабатывает свои параметры точно так же, как и функция `PQdescribePrepared`. Аналогично функции `PQdescribePrepared`, данная функция не будет работать при подключениях по протоколу версии 2.0.

`PQsendDescribePortal`

Отправляет запрос на получение информации об указанном портале и не дожидается завершения выполнения запроса.

```
int PQsendDescribePortal(PGconn *conn, const char *portalName);
```

Это асинхронная версия функции `PQdescribePortal`. Она возвращает 1, если ей удалось отправить запрос, и 0 в противном случае. После её успешного вызова следует вызвать функцию `PQgetResult` для получения результата. Эта функция обрабатывает свои параметры точно так же, как и функция `PQdescribePortal`. Аналогично функции `PQdescribePortal`, данная функция не будет работать при подключениях по протоколу версии 2.0.

`PQgetResult`

Ожидает получения следующего результата после предшествующего вызова `PQsendQuery`, `PQsendQueryParams`, `PQsendPrepare`, `PQsendQueryPrepared`, `PQsendDescribePrepared` или `PQsendDescribePortal` и возвращает его. Когда команда завершена и результатов больше не будет, тогда возвращается нулевой указатель.

```
PGresult *PQgetResult(PGconn *conn);
```

Функция `PQgetResult` должна вызываться повторно до тех пор, пока она не вернёт нулевой указатель, означающий, что команда завершена. (Если она вызвана, когда нет ни одной активной команды, тогда `PQgetResult` просто возвратит нулевой указатель сразу же.) Каждый ненулевой результат, полученный от `PQgetResult`, должен обрабатываться с помощью тех же самых функций доступа к структуре `PGresult`, которые были описаны выше. Не забывайте освобождать память, занимаемую каждым результирующим объектом, с помощью функции `PQclear`, когда работа с этим объектом закончена. Обратите внимание, что `PQgetResult` заблокируется, только если какая-либо команда активна, а необходимые ответные данные ещё не были прочитаны функцией `PQconsumeInput`.

> **Примечание**
>
> Даже когда `PQresultStatus` показывает фатальную ошибку, все равно следует вызывать функцию `PQgetResult` до тех пор, пока она не возвратит нулевой указатель, чтобы позволить libpq полностью обработать информацию об ошибке.

Использование `PQsendQuery` и `PQgetResult` решает одну из проблем `PQexec`: если строка команды содержит несколько SQL-команд, то результаты каждой из них можно получить индивидуально. (Между прочим, это позволяет организовать частичное совмещение обработки: клиент может обрабатывать результаты одной команды, в то время как сервер ещё работает с более поздними запросами, содержащимися в той же самой строке-команде.)

Ещё одной часто требующейся функциональной возможностью, которую можно получить с помощью `PQsendQuery` и `PQgetResult`, является извлечение больших выборок по одной строке за раз. Это обсуждается в Разделе 31.5.

Сам по себе вызов `PQgetResult` всё же заставит клиента заблокироваться до те пор, пока сервер не завершит выполнение следующей SQL-команды. Этого можно избежать с помощью надлежащего использования ещё двух функций:

`PQconsumeInput`

Если сервер готов передать данные, принять их.

```
int PQconsumeInput(PGconn *conn);
```

`PQconsumeInput` обычно возвращает 1, показывая, что «ошибки нет», но возвращает 0, если имела место какая-либо проблема (в таком случае можно обратиться к функции

PQerrorMessage за уточнением). Обратите внимание, что результат не говорит, были ли какие-либо входные данные фактически собраны. После вызова функции PQconsumeInput приложение может проверить PQisBusy и/или PQnotifies, чтобы посмотреть, не изменилось ли их состояние.

PQconsumeInput можно вызвать, даже если приложение ещё не готово иметь дело с результатом или уведомлением. Функция прочитает доступные данные и сохранит их в буфере, при этом обрабатывая условие готовности к чтению функции select(). Таким образом, приложение может использовать PQconsumeInput, чтобы немедленно обработать это состояние select(), а изучать результаты позже.

PQisBusy

Возвращает 1, если команда занята работой, то есть функция PQgetResult была бы заблокирована в ожидании ввода. Возвращаемое значение 0 показывает, что функция PQgetResult при её вызове гарантированно не будет заблокирована.

```
int PQisBusy(PGconn *conn);
```

Функция PQisBusy сама не будет пытаться прочитать данные с сервера; следовательно, сначала нужно вызвать PQconsumeInput, иначе состояние занятости никогда не прекратится.

В типичном приложении, использующем эти функции, будет главный цикл, в котором функции select() или poll() служат для организации ожидания наступления всех условий, на которые цикл должен отвечать. Одним из условий будет наличие ввода, доступного от сервера, что в терминах функции select() означает наличие данных, готовых для чтения на файловом дескрипторе, идентифицируемом с помощью PQsocket. Когда главный цикл обнаруживает ввод, готовый для чтения, он должен вызвать PQconsumeInput, чтобы прочитать этот ввод. Затем он может вызвать PQisBusy, а после неё уже PQgetResult, если PQisBusy возвратит false (0). Главный цикл может также вызвать PQnotifies, чтобы обнаружить сообщения NOTIFY (см. Раздел 31.8).

Клиент, который использует PQsendQuery/PQgetResult, может также попытаться отменить команду, которая всё ещё обрабатывается сервером; см. Раздел 31.6. Но, независимо от возвращаемого значения функции PQcancel, приложение должно продолжать обычную последовательность операций чтения результатов запроса, используя PQgetResult. Успешная отмена просто заставит команду завершиться раньше, чем она сделала бы в противном случае.

Используя функции, описанные выше, можно избежать блокирования, ожидая ввода от сервера баз данных. Однако всё же возможно, что приложение будет заблокировано в ожидании отправки вывода на сервер. Это бывает относительно нечасто, но может иметь место, если отправлены очень длинные SQL-команды или значения данных. (Однако это значительно более вероятно, если приложение отправляет данные через команду COPY IN.) Чтобы предотвратить эту возможность и достичь совершенно неблокирующего режима работы с базой данных, можно использовать следующие дополнительные функции.

PQsetnonblocking

Устанавливает неблокирующий статус подключения.

```
int PQsetnonblocking(PGconn *conn, int arg);
```

Устанавливает состояние подключения как неблокирующее, если *arg* равен 1, или блокирующее, если *arg* равен 0. Возвращает 0 в случае успешного завершения и -1 в случае ошибки.

В неблокирующем состоянии вызовы PQsendQuery, PQputline, PQputnbytes, PQputCopyData и PQendcopy не будут блокироваться, а вместо этого возвратят ошибку, если вызов должен быть повторным.

Обратите внимание, что функция PQexec не соблюдает неблокирующий режим. Если она вызывается, она всё равно работает в блокирующем режиме.

`PQisnonblocking`

Возвращает режим блокирования для подключения базы данных.

```
int PQisnonblocking(const PGconn *conn);
```

Возвращает 1, если подключение установлено в неблокирующем режиме, и 0, если режим блокирующий.

`PQflush`

Пытается сбросить любые выходные данные, стоящие в очереди, на сервер. Возвращает 0 в случае успеха (или если очередь на отправку пуста), -1 в случае сбоя по какой-либо причине или 1, если она ещё не смогла отправить все данные, находящиеся в очереди (этот случай может иметь место, только если соединение неблокирующее).

```
int PQflush(PGconn *conn);
```

После отправки любой команды или данных через неблокирующее подключение следует вызвать функцию `PQflush`. Если она возвратит 1, подождите, пока сокет станет готовым к чтению или записи. Если он станет готовым к записи, то вызовите `PQflush` снова. Если он станет готовым к чтению, то вызовите `PQconsumeInput`, а затем вызовите `PQflush` снова. Повторяйте до тех пор, пока `PQflush` не возвратит 0. (Необходимо выполнять проверку на состояние готовности к чтению и забирать входные данные с помощью `PQconsumeInput`, потому что сервер может заблокироваться, пытаясь отправить нам данные, например, сообщения NOTICE, и не будет читать наши данные до тех пор, пока мы не прочитаем его.) Как только `PQflush` возвратит 0, подождите, пока сокет не станет готовым к чтению, а затем прочитайте ответ, как описано выше.

31.5. Построчное извлечение результатов запроса

Обычно libpq собирает весь результат выполнения SQL-команды и возвращает его приложению в виде единственной структуры `PGresult`. Это может оказаться неприемлемым для команд, которые возвращают большое число строк. В таких случаях приложение может воспользоваться функциями `PQsendQuery` и `PQgetResult` в *однострочном режиме*. В этом режиме результирующие строки передаются приложению по одной за один раз, по мере того, как они принимаются от сервера.

Для того чтобы войти в однострочный режим, вызовите `PQsetSingleRowMode` сразу же после успешного вызова функции `PQsendQuery` (или родственной функции). Выбор этого режима действителен только для запроса, исполняемого в данный момент. Затем повторно вызывайте функцию `PQgetResult` до тех пор, пока она не возвратит null, как описано в Раздел 31.4. Если запрос возвращает какое-то число строк, то они возвращаются в виде индивидуальных объектов `PGresult`, которые выглядят, как обычные выборки, за исключением того, что их код статуса будет `PGRES_SINGLE_TUPLE` вместо `PGRES_TUPLES_OK`. После последней строки (или сразу же, если запрос не возвращает ни одной строки) будет возвращён объект, не содержащий ни одной строки и имеющий статус `PGRES_TUPLES_OK`; это сигнал о том, что строк больше не будет. (Но обратите внимание, что всё же необходимо продолжать вызывать функцию `PQgetResult`, пока она не возвратит значение null.) Все эти объекты `PGresult` будут содержать те же самые описательные данные (имена столбцов, типы и т. д.), которые имел бы обычный объект `PGresult`. Память, занимаемую каждым объектом, нужно освобождать с помощью `PQclear`, как обычно.

`PQsetSingleRowMode`

Выбирает однострочный режим для текущего выполняющегося запроса.

```
int PQsetSingleRowMode(PGconn *conn);
```

Эту функцию можно вызывать только непосредственно после функции `PQsendQuery` или одной из её родственных функций, до выполнения любой другой операции на этом подключении, такой, как `PQconsumeInput` или `PQgetResult`. Будучи вызванной своевременно, функция активирует однострочный режим для текущего запроса и возвращает 1. В противном случае режим остаётся не изменённым, а функция возвращает 0. В любом случае режим возвращается в нормальное состояние после завершения текущего запроса.

> **Внимание**
>
> В процессе обработки запроса сервер может возвратить некоторое количество строк, а затем столкнуться с ошибкой, вынуждающей его аварийно завершить запрос. Обычно libpq отбрасывает такие строки и сообщает только об ошибке. Но в однострочном режиме эти строки уже будут возвращены приложению. Следовательно, приложение увидит ряд объектов `PGresult`, имеющих статус `PGRES_SINGLE_TUPLE`, за которыми последует объект со статусом `PGRES_FATAL_ERROR`. Для обеспечения надлежащего поведения транзакций приложение должно быть спроектировано таким образом, чтобы отбрасывать или отменять все операции, проведённые с уже обработанными строками, если запрос в конечном итоге завершается сбоем.

31.6. Отмена запросов в процессе выполнения

Клиентское приложение может запросить отмену команды, которая ещё обрабатывается сервером, используя функции, описанные в этом разделе.

`PQgetCancel`

Создаёт структуру данных, содержащую информацию, необходимую для отмены команды, запущенной через конкретное подключение к базе данных.

```
PGcancel *PQgetCancel(PGconn *conn);
```

Функция `PQgetCancel` создаёт объект `PGcancel`, получив объект `PGconn`, описывающий подключение. Она возвратит `NULL`, если данный параметр *`conn`* равен `NULL` или представляет собой недействительное подключения. Объект `PGcancel` является непрозрачной структурой, которая не предназначена для того, чтобы приложение обращалось к ней напрямую; её можно только передавать функции `PQcancel` или `PQfreeCancel`.

`PQfreeCancel`

Освобождает память, занимаемую структурой данных, созданной функцией `PQgetCancel`.

```
void PQfreeCancel(PGcancel *cancel);
```

`PQfreeCancel` освобождает память, занимаемую объектом, предварительно созданным функцией `PQgetCancel`.

`PQcancel`

Требует, чтобы сервер прекратил обработку текущей команды.

```
int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);
```

Возвращаемое значение равно 1, если запрос на отмену был успешно отправлен, и 0 в противном случае. В случае неудачной отправки *`errbuf`* заполняется пояснительным сообщением об ошибке. *`errbuf`* должен быть массивом символов, имеющим размер *`errbufsize`* (рекомендуемый размер составляет 256 байт).

Успешная отправка ещё не является гарантией того, что запрос будет иметь какой-то эффект. Если отмена сработала, текущая команда завершится досрочно и возвратит в качестве результата ошибку. Если же отмена не получится (например, потому, что сервер уже завершил обработку команды), тогда вообще не будет видимого результата.

`PQcancel` можно безопасно вызывать из обработчика сигнала, если *`errbuf`* является локальной переменной в обработчике сигнала. Объект `PGcancel` доступен только в режиме чтения, пока речь идёт о функции `PQcancel`, поэтому её можно также вызывать из потока, отдельного от того, который управляет объектом `PGconn`.

`PQrequestCancel`

`PQrequestCancel` является нерекомендуемым вариантом функции `PQcancel`.

```
int PQrequestCancel(PGconn *conn);
```

Выдаёт запрос на то, чтобы сервер прекратил обработку текущей команды. Функция работает напрямую с объектом `PGconn` и в случае сбоя сохраняет сообщение об ошибке в объекте `PGconn` (откуда его можно извлечь с помощью `PQerrorMessage`). Хотя функциональность та же самая, этот подход создаёт риски для многопоточных программ и обработчиков сигналов, поскольку перезапись сообщения об ошибке, хранящегося в объекте `PGconn`, может помешать ходу операции, выполняемой через данное подключение.

31.7. Интерфейс быстрого пути

Postgres Pro предоставляет интерфейс для передачи серверу простых вызовов функций по быстрому пути.

> **Подсказка**
>
> Этот интерфейс несколько устарел, поскольку можно достичь подобной производительности и большей функциональности путём создания подготовленного оператора, определяющего вызов функции. Последующее выполнение этого оператора с передачей параметров и результатов в двоичном виде можно считать заменой вызову по быстрому пути.

Функция `PQfn`запрашивает выполнение серверной функции посредством интерфейса быстрого доступа:

```
PGresult *PQfn(PGconn *conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               const PQArgBlock *args,
               int nargs);

typedef struct
{
    int len;
    int isint;
    union
    {
        int *ptr;
        int integer;
    } u;
} PQArgBlock;
```

Аргумент *`fnid`* представляет собой OID функции, которая подлежит выполнению. *`args`* и *`nargs`* определяют параметры, которые должны быть переданы этой функции; они должны соответствовать списку аргументов объявленной функции. Когда поле *`isint`* структуры, передаваемой в качестве параметра, имеет значение true, тогда значение *`u.integer`* передаётся серверу в виде целого числа указанной длины (это должно быть 2 или 4 байта); при этом устанавливается нужный порядок байтов. Когда *`isint`* имеет значение false, тогда указанное число байт по адресу *`*u.ptr`* отправляется без какой-либо обработки; данные должны быть представлены в формате, которого ожидает сервер для передачи в двоичном виде данных того типа, что и аргументы функции. (Объявление поля *`u.ptr`*, как имеющего тип `int *`, является историческим; было бы лучше рассматривать его как тип `void *`.) *`result_buf`* указывает на буфер, в который должно быть помещено возвращаемое значение функции. Вызывающий код должен выделить достаточное место для сохранения возвращаемого значения. (Это никак не

проверяется!) Фактическая длина результирующего значения в байтах будет возвращена в переменной целого типа, на которую указывает *result_len*. Если ожидается получение двух- или четырёхбайтового целочисленного результата, то присвойте параметру *result_is_int* значение 1, в противном случае назначьте ему 0. Когда параметр *result_is_int* равен 1, libpq переставляет байты в передаваемом значении, если это необходимо, так, чтобы оно было доставлено на клиентскую машину в виде правильного значения типа int; обратите внимание, что по адресу **result_buf* доставляется четырёхбайтовое целое для любого допустимого размера результата. Когда *result_is_int* равен 0, тогда строка байтов в двоичном формате, отправленная сервером, будет возвращена немодифицированной. (В этом случае лучше рассматривать *result_buf* как имеющий тип void *.)

PQfn всегда возвращает действительный указатель на объект PGresult со статусом PGRES_COMMAND_OK при успешном выполнении функции или PGRES_FATAL_ERROR, если произошла какая-то ошибка. Перед использованием результата нужно сначала проверить его статус. Вызывающая функция отвечает за освобождение памяти, занимаемой объектом PGresult, когда он больше не нужен, с помощью PQclear.

Чтобы передать функции аргумент NULL, запишите в поле *len* этой структуры значение -1; поля *isint* и *u* в этом случае не важны. (Но это работает только при подключении по протоколу 3.0 и новее.)

Если функция возвращает NULL, в **result_len* записывается -1, а **result_buf* не изменяется. (Это работает только при подключении по протоколу 3.0 и новее; в протоколе 2.0 ни **result_len*, ни **result_buf* не изменяется.)

Обратите внимание, что при использовании этого интерфейса невозможно обработать множества значений в результате. Кроме того, функция должна быть обычной, а не агрегатной или оконной.

31.8. Асинхронное уведомление

Postgres Pro предлагает асинхронное уведомление посредством команд LISTEN и NOTIFY. Клиентский сеанс работы регистрирует свою заинтересованность в конкретном канале уведомлений с помощью команды LISTEN (и может остановить прослушивание с помощью команды UNLISTEN). Все сеансы, прослушивающие конкретный канал, будут уведомляться в асинхронном режиме, когда в рамках любого сеанса команда NOTIFY выполняется с параметром, указывающим имя этого канала. Для передачи дополнительных данных прослушивающим сеансам может использоваться строка «payload».

Приложения, использующие libpq, отправляют серверу команды LISTEN, UNLISTEN и NOTIFY, как обычные SQL-команды. Прибытие сообщений от команды NOTIFY можно впоследствии обнаружить с помощью вызова функции PQnotifies.

Функция PQnotifies возвращает следующее уведомление из списка необработанных уведомительных сообщений, полученных от сервера. Она возвращает нулевой указатель, если нет уведомлений, ожидающих обработки. Как только уведомление возвращено из функции PQnotifies, оно считается обработанным и будет удалено из списка уведомлений.

```
PGnotify *PQnotifies(PGconn *conn);

typedef struct pgNotify
{
    char *relname;              /* имя канала уведомлений */
    int  be_pid;                /* ID серверного процесса, посылающего уведомление */
    char *extra;                /* строка сообщения в уведомлении */
} PGnotify;
```

После обработки объекта PGnotify, возвращённого функцией PQnotifies, обязательно освободите память, занимаемую им, с помощью функции PQfreemem. Достаточно освободить указатель на PGnotify; поля relname и extra не представляют отдельных областей памяти. (Имена этих полей

являются таковыми по историческим причинам; в частности, имена каналов не обязаны иметь ничего общего с именами реляционных отношений.)

Пример 31.2 представляет пример программы, иллюстрирующей использование асинхронного уведомления.

Функция PQnotifies в действительности не читает данные с сервера; она просто возвращает сообщения, предварительно собранные другой функцией библиотеки libpq. В очень старых выпусках libpq обеспечить своевременное получения сообщений от команды NOTIFY можно было только одним способом — постоянно отправлять команды, пусть даже пустые, а затем проверять PQnotifies после каждого вызова PQexec. Хотя этот метод всё ещё работает, он считается устаревшим ввиду неэффективного использования процессора.

Более удачным способом проверки наличия сообщений от команды NOTIFY, когда у вас нет полезных команд для выполнения, является вызов функции PQconsumeInput с последующей проверкой PQnotifies. Вы можете использовать select(), чтобы подождать прибытия данных с сервера, не занимая тем самым ресурсы CPU в отсутствие задач для выполнения. (Получить номер дескриптора для использования с select() можно с помощью функции PQsocket.) Заметьте, что это будет работать независимо от того, отправляете ли вы команды, используя PQsendQuery/PQgetResult, или просто вызываете PQexec. Однако важно не забывать проверять PQnotifies после каждого вызова PQgetResult или PQexec, чтобы увидеть, не поступили ли какие-либо уведомления в процессе обработки команды.

31.9. Функции, связанные с командой COPY

Команда COPY в Postgres Pro имеет возможность читать и записывать данные через сетевое подключение, установленное libpq. Описанные в этом разделе функции позволяют приложениям воспользоваться этой возможностью для передачи или приёма копируемых данных.

Общая процедура такова: сначала приложение выдаёт SQL-команду COPY, вызывая PQexec или одну из подобных функций. В ответ оно должно получить (если не возникла ошибка) объект PGresult с кодом состояния PGRES_COPY_OUT или PGRES_COPY_IN (в зависимости от направления копирования). Затем приложение должно использовать функции, описанные в этом разделе, и принимать или передавать строки данных. По завершении передачи возвращается ещё один объект PGresult, сообщающий о состоянии завершения передачи. В случае успеха он содержит код состояния PGRES_COMMAND_OK, а если возникает какая-то проблема — PGRES_FATAL_ERROR. После этого можно продолжать выполнять SQL-команды через PQexec. (Пока операция COPY не завершена, выполнять другие SQL-команды через то же подключение нельзя.)

Если команда COPY была выполнена через PQexec в строке, содержащей дополнительные команды, приложение должно продолжить получать результаты через PQgetResult после завершения последовательности COPY. Только когда PQgetResult возвращает NULL, можно с уверенностью считать, что командная строка PQexec выполнена полностью, и безопасно передавать другие команды.

Функции, описанные в этом разделе, должны выполняться только после получения кода состояния PGRES_COPY_OUT или PGRES_COPY_IN от функции PQexec или PQgetResult.

Объект PGresult с таким кодом состояния содержит дополнительные данные о начавшейся операции COPY. Эти данные можно получить функциями, также применяющимися при обработке результатов запроса:

PQnfields

Возвращает число копируемых столбцов (полей).

PQbinaryTuples

Значение 0 указывает, что для всей операции копирования применяется текстовый формат (строки разделяются символами новой строки, столбцы разделяются символами-

разделителями и т. д.). Значение 1 указывает, что для всей операции копирования применяется двоичный формат. За дополнительными сведениями обратитесь к COPY.

`PQfformat`

Возвращает код формата (0 — текстовый, 1 — двоичный), связанный с каждым копируемым столбцом. Коды форматов столбцов всегда будут нулевыми, если общий формат копирования — текстовый, но с двоичным форматом поддерживаются и текстовые, и двоичные столбцы. (Однако в текущей реализации `COPY` при двоичном копировании столбцы могут быть только двоичными, так что форматы столбцов должны всегда соответствовать общему формату.)

> **Примечание**
>
> Эти дополнительные значения данных доступны только при использовании протокола 3.0. С протоколом 2.0 все эти функции возвращают 0.

31.9.1. Функции для передачи данных `COPY`

Эти функции применяются для передачи данных при операции `COPY FROM STDIN`. Они не будут работать, если подключение находится не в состоянии `COPY_IN`.

`PQputCopyData`

Отправляет данные на сервер, когда активно состояние `COPY_IN`.

```
int PQputCopyData(PGconn *conn,
                  const char *buffer,
                  int nbytes);
```

Передаёт серверу данные `COPY` из указанного буфера (*`buffer`*), длиной *`nbytes`* байт. Она возвращает 1, если данные были переданы, 0, если они не попали в очередь, так как буферы были заполнены (это возможно только в неблокирующем режиме), или -1, если произошла ошибка. (Если возвращено -1, подробности ошибки можно узнать, вызвав `PQerrorMessage`. Если получен 0, дождитесь состояния готовности к записи и повторите попытку.)

Приложение может разделять поток данных `COPY` на буферизуемые блоки любого удобного размера. Границы буфера не имеют семантического значения при передаче. Содержимое потока данных должно соответствовать формату данных, ожидаемому командой `COPY`; за подробностями обратитесь к COPY.

`PQputCopyEnd`

Отправляет признак конца данных на сервер, когда активно состояние `COPY_IN`.

```
int PQputCopyEnd(PGconn *conn,
                 const char *errormsg);
```

Завершает операцию `COPY_IN` с успешным результатом, если в *`errormsg`* передаётся `NULL`. Если *`errormsg`* не `NULL`, команда `COPY` будет завершена с ошибкой, а сообщением об ошибке будет строка, переданная в *`errormsg`*. (Однако не следует полагать, что именно это сообщение будет получено от сервера назад, так как сервер мог уже прервать операцию `COPY` по своим причинам. Также заметьте, что принудительный вызов ошибки не работает с соединениями по протоколу версии до 3.0.)

Эта функция возвращает 1, если сообщение завершения было передано; в неблокирующем режиме это означает только, что сообщение завершения успешно поставлено в очередь. (Чтобы удостовериться, что данные были успешно отправлены в неблокирующем режиме, следует дождаться готовности к записи и вызывать `PQflush` в цикле, пока она не вернёт ноль.) Нулевой результат означает, что функция не смогла поставить сообщение завершения

в очередь по причине заполнения буферов; это возможно только в неблокирующем режиме. (В этом случае нужно дождаться готовности к записи и попытаться вызвать PQputCopyEnd снова.) Если действительно происходит ошибка, возвращается -1; получить её подробности можно, вызвав PQerrorMessage.

После успешного вызова PQputCopyEnd вызовите PQgetResult, чтобы узнать окончательный результат команды COPY. Ожидать появления этого результата можно обычным образом. Затем вернитесь к обычным операциям.

31.9.2. Функции для приёма данных COPY

Эти функции применяются для получения данных при операции COPY TO STDOUT. Они не будут работать, если подключение находится не в состоянии COPY_OUT.

PQgetCopyData

Принимает данные от сервера, когда активно состояние COPY_OUT.

```
int PQgetCopyData(PGconn *conn,
                  char **buffer,
                  int async);
```

Запрашивает следующую строку данных с сервера в процессе операции COPY. Данные всегда возвращаются строка за строкой; если поступила только часть строки, она не возвращается. Успешное получение строки данных подразумевает выделение блока памяти для этих данных. В параметре *buffer* ей передаётся указатель, отличный от NULL. По адресу **buffer* записывается указатель на выделенную память, либо NULL, когда буфер не возвращается. Если буфер результата отличен от NULL, его следует освободить, когда он станет не нужен, вызвав PQfreemem.

Когда строка получена успешно, возвращается число байт данных в этой строке (это число всегда больше нуля). Возвращаемое строковое значение всегда завершается нулём, хотя это полезно, вероятно, только для текстовой COPY. Нулевой результат означает, что операция COPY продолжает выполняться, но строка ещё не готова (это возможно, только когда параметр *async* равен true). Возвращённое значение -1 означает, что команда COPY завершена, а -2 показывает, что произошла ошибка (её причину можно узнать с помощью PQerrorMessage).

Когда параметр *async* равен true (отличен от нуля), функция PQgetCopyData не будет блокироваться, ожидая данных; она возвратит ноль, если выполнение COPY продолжается, но полная строка ещё не получена. (В этом случае нужно дождаться готовности к чтению и затем вызвать PQconsumeInput, прежде чем вызывать PQgetCopyData ещё раз.) Когда *async* равен false (нулю), PQgetCopyData будет заблокирована до поступления данных или окончания операции.

Когда PQgetCopyData возвращает -1, вызовите PQgetResult, чтобы узнать окончательный результат команды COPY. Ожидать появления этого результата можно обычным образом. Затем вернитесь к обычным операциям.

31.9.3. Устаревшие функции для COPY

Эти функции представляют старые методы выполнения операции COPY. Хотя они продолжают работать, они признаны устаревшими из-за плохой обработки ошибок, неудобных способов обнаружения конца данных и отсутствия поддержки двоичных или неблокирующих передач.

PQgetline

Читает передаваемую сервером строку символов, завершающуюся символом новой строки, в буфер (buffer) размера *length*.

```
int PQgetline(PGconn *conn,
              char *buffer,
```

```
                int length);
```

Эта функция копирует *length*-1 символов в буфер и преобразует символ конца строки в нулевой байт. PQgetline возвращает EOF в конце ввода, 0, если была прочитана вся строка, и 1, если буфер заполнен, но завершающий символ конца строки ещё не прочитан.

Заметьте, что приложение должно проверить, не состоит ли новая строка в точности из двух символов \., что будет означать, что сервер завершил передачу результатов команды COPY. Если приложение может принимать строки длиннее *length*-1 символов, необходимо позаботиться о том, чтобы оно корректно распознавало строку \. (а не воспринимало, например, конец длинной строки данных как завершающую строку).

PQgetlineAsync

Читает передаваемую сервером строку данных COPY в буфер без блокировки.

```
int PQgetlineAsync(PGconn *conn,
                   char *buffer,
                   int bufsize);
```

Эта функция похожа на PQgetline, но может применяться в приложениях, которые должны читать данные COPY асинхронно, то есть, без блокировки. Запустив команду COPY и получив ответ PGRES_COPY_OUT, приложение должно вызывать PQconsumeInput и PQgetlineAsync, пока не будет получен сигнал конца данных.

В отличие от PQgetline, эта функция сама отвечает за обнаружение конца данных.

При каждом вызове PQgetlineAsync будет возвращать данные, если во входном буфере libpq оказывается полная строка данных. В противном случае никакие данные не возвращаются до поступления остального содержимого строки. Эта функция возвращает -1, если обнаруживается признак завершения копирования, или 0, если данные не получены, или положительное количество возвращённых байт данных. Если возвращается -1, вызывающий код должен затем вызвать PQendcopy и после этого перейти в обычный режим работы.

Возвращаемые данные не будут пересекать границы строк данных. При этом может быть возвращена одна строка целиком. Но если буфер, выделенный вызывающим кодом, оказывается слишком мал для строки, передаваемой сервером, возвращена будет часть строки. Когда передаются текстовые данные, это можно выявить, проверив, содержит ли последний возвращаемый байт символ \n. (Для COPY в двоичном формате потребуется собственно разобрать формат данных COPY, чтобы выявить подобную ситуацию.) Возвращаемая строка не завершается нулём. (Если вы хотите получить строку с нулём в конце, передайте в *bufsize* число на единицу меньше фактического размера блока.)

PQputline

Передаёт серверу строку, завершённую нулём. Возвращает 0 в случае успеха, либо EOF, если передать строку не удаётся.

```
int PQputline(PGconn *conn,
              const char *string);
```

Поток данных COPY, передаваемых последовательностью вызовов PQputline, имеет тот же формат, что возвращает PQgetlineAsync, за исключением того, что приложения не обязательно должны передавать по одной строке данных за вызов PQputline; они могут посылать части строк или сразу несколько строк.

> **Примечание**
>
> До версии 3.0 протокола Postgres Pro приложение должно было явно отправлять два символа \. последней строкой, чтобы сообщить серверу, что оно закончило передачу

данных COPY. Хотя это по-прежнему работает, такое поведение считается устаревшим и ожидается, что особое значение \. будет исключено в будущих версиях. Передав собственно данные, сейчас достаточно вызвать PQendcopy.

PQputnbytes

Передаёт серверу строку, не завершённую нулём. Возвращает 0 в случае успеха, либо EOF, если передать строку не удаётся.

```
int PQputnbytes(PGconn *conn,
                const char *buffer,
                int nbytes);
```

Поведение этой функции не отличается от PQputline, но её буфер данных не должен содержать завершающий ноль, так как для неё число передаваемых байт задаётся непосредственно. Используйте эту функцию для передачи двоичных данных.

PQendcopy

Производит синхронизацию с сервером.

```
int PQendcopy(PGconn *conn);
```

Эта функция ожидает завершения копирования сервером. Её следует вызывать, либо когда серверу была передана последняя строка функцией PQputline, либо когда от сервера была получена последняя строка функцией PQgetline. Если её не вызвать, сервер «потеряет синхронизацию» с клиентом. После завершения этой функции сервер готов принимать следующую команду SQL. В случае успешного завершения возвращается 0, в противном случае — ненулевое значение. (Чтобы получить подробности ошибки при ненулевом значении, вызовите PQerrorMessage.)

Вызывая PQgetResult, приложение должно обрабатывать результат PGRES_COPY_OUT, в цикле выполняя PQgetline, а обнаружив завершающую строку, вызвать PQendcopy. Затем оно должно вернуться к циклу PQgetResult, и выйти из него, когда PQgetResult возвратит нулевой указатель. Подобным образом, получив результат PGRES_COPY_IN, приложение должно выполнить серию вызовов PQputline, завершить её, вызвав PQendcopy, а затем вернуться к циклу PQgetResult. При такой организации обработки команда COPY будет корректно выполняться и в составе последовательности команд SQL.

Старые приложения обычно передают команду COPY через PQexec и рассчитывают, что транзакция будет завершена после PQendcopy. Это будет работать, только если команда COPY является единственной SQL-командой в строке команд.

31.10. Функции управления

Эти функции управляют различными аспектами поведения libpq.

PQclientEncoding

Возвращает кодировку клиента.

```
int PQclientEncoding(const PGconn *conn);
```

Заметьте, что она возвращает идентификатор кодировки, а не символьную строку вида EUC_JP. В случае ошибки она возвращает -1. Преобразовать идентификатор кодировки в имя можно, воспользовавшись следующей функцией:

```
char *pg_encoding_to_char(int encoding_id);
```

PQsetClientEncoding

Устанавливает кодировку клиента.

```
int PQsetClientEncoding(PGconn *conn, const char *encoding);
```

В *conn* передаётся соединение с сервером, а в *encoding* — имя требуемой кодировки. Если функция устанавливает кодировку успешно, она возвращает 0, или -1 в противном случае. Определить текущую кодировку для соединения можно, воспользовавшись функцией PQclientEncoding.

PQsetErrorVerbosity

Определяет уровень детализации сообщений, возвращаемых функциями PQerrorMessage и PQresultErrorMessage.

```
typedef enum
{
    PQERRORS_TERSE,
    PQERRORS_DEFAULT,
    PQERRORS_VERBOSE
} PGVerbosity;

PGVerbosity PQsetErrorVerbosity(PGconn *conn, PGVerbosity verbosity);
```

PQsetErrorVerbosity устанавливает уровень детализации и возвращает предыдущее значение для соединения. В «лаконичном» режиме (*TERSE*) возвращаемые сообщения содержат только уровень важности, основной текст и позицию; всё это обычно умещается в одной строке. В режиме по умолчанию выдаваемые сообщения дополнительно содержат поля подробного описания, подсказки или контекста (они могут занимать несколько строк). В «многословном» режиме (*VERBOSE*) передаются все доступные поля сообщения. Изменение уровня детализации не влияет на сообщения, уже сформированные в существующих объектах PGresult, а затрагивает только последующие сообщения. (Но можно воспользоваться PQresultVerboseErrorMessage, чтобы получить предыдущую ошибку с другим уровнем детализации.)

PQsetErrorContextVisibility

Определяет вариант обработки полей CONTEXT в сообщениях, возвращаемых функциями PQerrorMessage и PQresultErrorMessage.

```
typedef enum
{
    PQSHOW_CONTEXT_NEVER,
    PQSHOW_CONTEXT_ERRORS,
    PQSHOW_CONTEXT_ALWAYS
} PGContextVisibility;

PGContextVisibility PQsetErrorContextVisibility(PGconn *conn, PGContextVisibility
 show_context);
```

PQsetErrorContextVisibility устанавливает режим вывода контекста сообщений и возвращает предыдущее значение. В зависимости от этого режима, поле CONTEXT может включаться в сообщения или нет (если только не выбран уровень детализации *TERSE*, при котором CONTEXT не выводится никогда). В режиме *NEVER* поле CONTEXT не выводится никогда, а в режиме *ALWAYS* выводится всегда (если оно имеется). В режиме *ERRORS* (по умолчанию) поле CONTEXT включается только в сообщения об ошибках, но не в уведомления или предупреждения. Изменение этого уровня не влияет на сообщения, уже сформированные в существующих объектах PGresult, а затрагивает только последующие сообщения. (Но можно воспользоваться PQresultVerboseErrorMessage, чтобы получить предыдущую ошибку с другим режимом вывода контекста.)

PQtrace

Включает трассировку клиент-серверного взаимодействия с выводом в поток отладочных сообщений.

```
void PQtrace(PGconn *conn, FILE *stream);
```

> **Примечание**
>
> В Windows, если библиотека libpq и приложение скомпилированы с разными флагами, эта функция может вызвать крах приложения из-за различий внутреннего представления указателей `FILE`. В частности, флаги многопоточной/однопоточной, выпускаемой/отладочной или статической/динамической сборки должны быть одинаковыми для библиотеки и всех использующих её приложений.

`PQuntrace`

Выключает трассировку, запущенную функцией `PQtrace`.

```
void PQuntrace(PGconn *conn);
```

31.11. Функции разного назначения

Как всегда, находятся функции, которые не попадают ни в одну из категорий.

`PQfreemem`

Освобождает память, которую выделила libpq.

```
void PQfreemem(void *ptr);
```

Освобождает память, выделенную библиотекой libpq, а именно функциями `PQescapeByteaConn`, `PQescapeBytea`, `PQunescapeBytea` и `PQnotifies`. Особенно важно использовать именно эту функцию, а не `free()`, в Microsoft Windows. Это связано с тем, что выделение памяти в DLL и освобождение её в приложении будет работать, только если флаги многопоточной/однопоточной, выпускаемой/отладочной или статической/динамической сборки для DLL и приложения полностью совпадают. На других платформах эта функция действует так же, как стандартная библиотечная функция `free()`.

`PQconninfoFree`

Освобождает структуры данных, выделенные функциями `PQconndefaults` и `PQconninfoParse`.

```
void PQconninfoFree(PQconninfoOption *connOptions);
```

Простая функция `PQfreemem` не подойдёт для этого, так как эти структуры содержат ссылки на подчинённые строки.

`PQencryptPasswordConn`

Подготавливает зашифрованную форму пароля Postgres Pro.

```
char *PQencryptPasswordConn(PGconn *conn, const char *passwd, const char *user,
 const char *algorithm);
```

Эта функция предназначена для клиентских приложений, желающих передавать команды вида `ALTER USER joe PASSWORD 'pwd'`. В такой команде лучше не передавать исходный пароль открытым текстом, так как он может появиться в журналах команд, мониторе активности и т. д. Вместо этого воспользуйтесь данной функцией и переведите пароль в зашифрованную форму.

В аргументах *`passwd`* и *`user`* задаётся пароль в открытом виде и SQL-имя пользователя, для которого он задаётся. В аргументе *`algorithm`* задаётся алгоритм для шифрования пароля. В настоящее время поддерживаются алгоритмы `md5` и `scram-sha-256` (в качестве альтернативных обозначений `md5` для совместимости со старыми версиями сервера поддерживаются значения `on` и `off`). Заметьте, что поддержка `scram-sha-256` появилась в Postgres Pro версии 10 и

со старыми версиями серверов этот вариант работать не будет. Если *algorithm* равен NULL, эта функция запросит у сервера текущее значение параметра password_encryption. При этом возможна блокировка и отказ при выполнении функции, если текущая транзакция прерывается или если в текущем соединении выполняется другой запрос. Если вы хотите использовать алгоритм по умолчанию для данного сервера, но при этом избежать блокировки, получите значение password_encryption самостоятельно до вызова PQencryptPasswordConn и передайте его в параметре *algorithm*.

Эта функция возвращает строку, выделенную функцией malloc. Вызывающий код может рассчитывать на то, что эта строка не содержит никаких спецсимволов, требующих экранирования. Закончив работу с ней, освободите память, вызвав PQfreemem. В случае ошибки эта функция возвращает NULL, а соответствующее сообщение помещается в объект соединения.

PQencryptPassword

Подготавливает зашифрованную md5 форму пароля Postgres Pro.

```
char *PQencryptPassword(const char *passwd, const char *user);
```

PQencryptPassword — старая, подлежащая ликвидации версия PQencryptPasswordConn. Отличие состоит в том, что для PQencryptPassword не требуется объект соединения, а в качестве алгоритма шифрования всегда используется md5.

PQmakeEmptyPGresult

Конструирует пустой объект PGresult с указанным состоянием.

```
PGresult *PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
```

Это внутренняя функция libpq, выделяющая память и инициализирующая пустой объект PGresult. Эта функция возвращает NULL, если не может выделить память. Она сделана экспортируемой, так как некоторые приложения находят полезным создавать объекты результатов (в частности, объекты с состоянием ошибки) самостоятельно. Если в *conn* передаётся не null и *status* указывает на ошибку, в PGresult копируется текущее сообщение об ошибке для заданного соединения. Также, если в *conn* передаётся не null, в PGresult копируются все процедуры событий, зарегистрированные для этого соединения. (При этом вызовы PGEVT_RESULTCREATE не выполняются; см. описание PQfireResultCreateEvents.) Заметьте, что в конце для этого объекта следует вызвать PQclear, как и для объекта PGresult, возвращённого самой библиотекой libpq.

PQfireResultCreateEvents

Вызывает событие PGEVT_RESULTCREATE (см. Раздел 31.13) для каждой процедуры событий, зарегистрированной в объекте PGresult. Возвращает ненулевое значение в случае успеха или ноль в случае ошибки в одной из процедур.

```
int PQfireResultCreateEvents(PGconn *conn, PGresult *res);
```

Аргумент conn передаётся процедурам событий, но непосредственно не используется. Он может быть равен NULL, если он не нужен процедурам событий.

Процедуры событий, уже получившие событие PGEVT_RESULTCREATE или PGEVT_RESULTCOPY для этого объекта, больше не вызываются.

Основная причина отделения этой функции от PQmakeEmptyPGresult в том, что часто требуется создать объект PGresult и наполнить его данными, прежде чем вызывать процедуры событий.

PQcopyResult

Создаёт копию объекта PGresult. Эта копия никак не связана с исходным результатом и поэтому, когда она становится не нужна, необходимо вызвать PQclear. Если функция завершается ошибкой, она возвращает NULL.

```
PGresult *PQcopyResult(const PGresult *src, int flags);
```

Создаваемая копия не будет точной. В возвращаемый результат всегда помещается состояние `PGRES_TUPLES_OK` и в него не копируются никакие сообщения об ошибках из исходного объекта. (Однако в него копируется строка состояния команды.) Что ещё в него будет копироваться, определяет аргумент *`flags`*, в котором складываются несколько флагов. Флаг `PG_COPYRES_ATTRS` включает копирование атрибутов исходного объекта (определений столбцов), а флаг `PG_COPYRES_TUPLES` включает копирование кортежей из исходного объекта (при этом также копируются и атрибуты.) Флаг `PG_COPYRES_NOTICEHOOKS` включает копирование обработчиков замечаний, а флаг `PG_COPYRES_EVENTS` — событий из исходного объекта результата. (Но любые данные, связанные с экземпляром исходного объекта, не копируются.)

`PQsetResultAttrs`

Устанавливает атрибуты объекта `PGresult`.

```
int PQsetResultAttrs(PGresult *res, int numAttributes, PGresAttDesc *attDescs);
```

Предоставленная структура *`attDescs`* копируется в результат. Если указатель *`attDescs`* равен `NULL` или *`numAttributes`* меньше одного, запрос игнорируется и функция выполняется без ошибки. Если *`res`* уже содержит атрибуты, функция завершается ошибкой. В случае ошибки функция возвращает ноль, а в обратном случае — ненулевое значение.

`PQsetvalue`

Устанавливает значение поля кортежа в объекте `PGresult`.

```
int PQsetvalue(PGresult *res, int tup_num, int field_num, char *value, int len);
```

Эта функция автоматически увеличивает внутренний массив кортежей при необходимости. Однако значение *`tup_num`* должно быть меньше или равно `PQntuples`, что означает, что эта функция может увеличивать массив кортежей только на один кортеж. Но в существующем кортеже любые поля могут изменяться в любом порядке. Если значение в поле с номером *`field_num`* уже существует, оно будет перезаписано. Если *`len`* равно -1 или *`value`* равно `NULL`, в поле будет записано значение SQL NULL. Устанавливаемое значение (*`value`*) копируется в закрытую область объекта результата, так что от него можно избавиться после завершения функции. Если функция завершается ошибкой, она возвращает ноль, а в обратном случае — ненулевое значение.

`PQresultAlloc`

Выделяет подчинённую область памяти для объекта `PGresult`.

```
void *PQresultAlloc(PGresult *res, size_t nBytes);
```

Любая память, выделенная этой функцией, будет освобождена при очистке объекта *`res`*. В случае ошибки эта функция возвращает `NULL`. Результат гарантированно выравнивается должным образом для любого типа данных, как и при `malloc`.

`PQlibVersion`

Возвращает версию используемой библиотеки libpq.

```
int PQlibVersion(void);
```

По результату этой функции можно во время выполнения определить, предоставляется ли определённая функциональность загруженной в данный момент версией libpq. Эта функция может использоваться, например, чтобы понять, какие параметры соединения может принять `PQconnectdb`.

Это число формируется в результате умножения номера основной версии библиотеки на 10000 и добавления номера дополнительной версии. Например, для версии 10.1 будет выдано 100001, а для версии 11.0 — 110000.

До версии 10, в Postgres Pro номера версий образовывались из трёх чисел, первые два из которых представляли основную версию. Для таких версий `PQlibVersion` отводит на каждое число по две цифры; например, для версии 9.1.5 будет выдано 90105, а для версии 9.2.0 — 90200.

Таким образом, чтобы получить логический номер версии для определения доступности функционала, приложения должны разделить результат `PQlibVersion` на 100, а не на 10000. Во всех сериях номера дополнительных (корректирующих) выпусков различаются только в двух последних цифрах.

> **Примечание**
>
> Эта функция появилась в PostgreSQL версии 9.1, поэтому с её помощью нельзя проверить функциональность предыдущих версий, так как при вызове её будет создана зависимость от версии 9.1 или новее.

31.12. Обработка замечаний

Сообщения с замечаниями и предупреждениями, выдаваемые сервером, не возвращаются функциями, выполняющими запросы, так как они не свидетельствуют об ошибке в запросе. Вместо этого они передаются функции обработки замечаний и после завершения этой функции выполнение продолжается как обычно. Стандартная функция обработки замечаний выводит сообщение в `stderr`, но приложение может переопределить это поведение, предоставив собственный обработчик.

По историческим причинам обработка замечаний выполняется на двух уровнях, приёмником замечаний и обработчиком замечаний. По умолчанию приёмник замечаний форматирует замечание и передаёт сформированную строку обработчику замечаний для вывода. Однако приложения, которые реализуют свой приёмник замечаний, обычно просто игнорируют слой обработчика и выполняют все действия в коде приёмника.

Функция `PQsetNoticeReceiver` устанавливает или возвращает текущий приёмник замечаний для объекта соединения. Подобным образом, `PQsetNoticeProcessor` устанавливает или возвращает текущий обработчик замечаний.

```
typedef void (*PQnoticeReceiver) (void *arg, const PGresult *res);

PQnoticeReceiver
PQsetNoticeReceiver(PGconn *conn,
                    PQnoticeReceiver proc,
                    void *arg);

typedef void (*PQnoticeProcessor) (void *arg, const char *message);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                     PQnoticeProcessor proc,
                     void *arg);
```

Каждая из этих функций возвращает указатель на ранее установленный приёмник или обработчик замечаний и устанавливает новый указатель. Если ей передаётся нулевой указатель, она ничего не делает, только возвращает текущий указатель.

Когда сообщение с замечанием или предупреждением поступает от сервера, либо выдаётся самой библиотекой libpq, вызывается функция приёмника замечания. Сообщение передаётся ей в виде состояния `PGRES_NONFATAL_ERROR` объекта `PGresult`. (Это позволяет приёмнику извлечь из него отдельные поля, используя `PQresultErrorField`, либо получить полное готовое сообщение, вызвав `PQresultErrorMessage` или `PQresultVerboseErrorMessage`.) Ей также передаётся тот же

неопределённый указатель, что был передан функции PQsetNoticeReceiver. (Этот указатель может пригодиться для обращения к внутреннему состоянию приложения при необходимости.)

Стандартный приёмник замечаний просто извлекает сообщение (вызывая PQresultErrorMessage) и передаёт его обработчику замечаний.

Обработчик замечаний отвечает за обработку сообщения с замечанием или предупреждением в текстовом виде. Ему передаётся строка с текстом сообщения (включающая завершающий символ новой строки) и неопределённый указатель, который был передан функции PQsetNoticeProcessor. (Этот указатель может пригодиться для обращения к внутреннему состоянию приложения при необходимости.)

Стандартный обработчик замечаний прост:

```
static void
defaultNoticeProcessor(void *arg, const char *message)
{
    fprintf(stderr, "%s", message);
}
```

Установив приёмник или обработчик замечаний, вы можете ожидать, что эта функция будет вызываться, пока будут существовать объект PGconn или объекты PGresult, созданные с ней. Когда создаётся PGresult, указатели текущих обработчиков замечаний, установленные в PGconn, копируются в PGresult для возможного использования функциями вроде PQgetvalue.

31.13. Система событий

Система событий libpq разработана для уведомления функций-обработчиков об интересных событиях libpq, например, о создании и уничтожении объектов PGconn и PGresult. Основное их предназначение в том, чтобы позволить приложениям связать собственные данные с объектами PGconn и PGresult и обеспечить их освобождение в нужное время.

Каждый зарегистрированный обработчик событий связывается с двумя элементами данных, которые известны libpq только как скрытые указатели void *. Первый *сквозной* указатель передаётся приложением, когда обработчик событий регистрируется в PGconn. Этот указатель никогда не меняется на протяжении жизни PGconn и все объекты PGresult создаются с ним; поэтому, если он используется, он должен указывать на долгоживущие данные. В дополнение к нему имеется указатель *данных экземпляра*, который изначально равен NULL во всех объектах PGconn и PGresult. Этим указателем можно управлять с помощью функций PQinstanceData, PQsetInstanceData, PQresultInstanceData и PQsetResultInstanceData. Заметьте, что в отличие от сквозного указателя, данные экземпляра PGconn автоматически не наследуются объектами PGresult, создаваемыми из него. Библиотека libpq не знает, на что указывают сквозной указатель и указатель данных экземпляра (если они ненулевые) и никогда не будет пытаться освобождать их — за это отвечает обработчик событий.

31.13.1. Типы событий

Перечисление PGEventId описывает типы событий, обрабатываемых системой событий. Имена всех их значений начинаются с PGEVT. Для каждого типа событий имеется соответствующая структура информации о событии, содержащая параметры, передаваемые обработчикам событий. Определены следующие типы событий:

PGEVT_REGISTER

Событие регистрации происходит, когда вызывается PQregisterEventProc. Это идеальное время для инициализации данных экземпляра (instanceData), которые могут понадобиться процедуре событий. Для каждого обработчика событий в рамках соединения будет выдаваться только одно событие регистрации. Если обработка события завершается ошибкой, регистрация прерывается.

```
typedef struct
{
    PGconn *conn;
} PGEventRegister;
```

При поступлении события PGEVT_REGISTER указатель *evtInfo* следует привести к PGEventRegister *. Эта структура содержит объект PGconn, который должен быть в состоянии CONNECTION_OK; это гарантируется, если PQregisterEventProc вызывается сразу после получения рабочего объекта PGconn. В случае выдачи кода ошибки всю очистку необходимо провести самостоятельно, так как событие PGEVT_CONNDESTROY не поступит.

PGEVT_CONNRESET

Событие сброса соединения происходит при завершении PQreset или PQresetPoll. В обоих случаях это событие вызывается, только если сброс был успешным. Если обработка события завершается ошибкой, происходит сбой всей операции сброса соединения; объект PGconn переходит в состояние CONNECTION_BAD и PQresetPoll возвращает PGRES_POLLING_FAILED.

```
typedef struct
{
    PGconn *conn;
} PGEventConnReset;
```

При поступлении события PGEVT_CONNRESET указатель *evtInfo* следует привести к PGEventConnReset *. Хотя переданный объект PGconn был только что сброшен, все данные события остаются неизменными. При поступлении этого события должны быть сброшены/перезагружены/вновь запрошены все сопутствующие данные instanceData. Заметьте, что даже если обработчик события выдаст ошибку при обработке PGEVT_CONNRESET, событие PGEVT_CONNDESTROY всё равно поступит при закрытии соединения.

PGEVT_CONNDESTROY

Событие уничтожения соединения вызывается в ответ на вызов PQfinish. Обработчик этого события отвечает за корректную очистку своих данных событий, так как libpq не может управлять его памятью. Невыполнение очистки должным образом приведёт к утечкам памяти.

```
typedef struct
{
    PGconn *conn;
} PGEventConnDestroy;
```

При поступлении события PGEVT_CONNDESTROY указатель *evtInfo* следует привести к PGEventConnDestroy *. Это событие происходит перед тем, как PQfinish производит всю остальную очистку. Значение, возвращаемое обработчиком событий, игнорируется, так как из PQfinish никак нельзя сообщить об ошибке. Кроме того, ошибка в обработчике событий не должна прерывать процесс очистки ставшей ненужной памяти.

PGEVT_RESULTCREATE

Событие создания объекта результата происходит при завершении любой функции, выполняющей запрос и получающей результат, включая PQgetResult. Это событие происходит только после того, как результат был успешно получен.

```
typedef struct
{
    PGconn *conn;
    PGresult *result;
} PGEventResultCreate;
```

При поступлении события PGEVT_RESULTCREATE указатель *evtInfo* следует привести к PGEventResultCreate *. В *conn* передаётся соединение, для которого сформирован результат. Это идеальное место для инициализации любых данных instanceData, которые нужно

связать с результатом. В случае сбоя обработчика объект результата очищается и ошибка распространяется дальше. Обработчик события не должен пытаться выполнять PQclear для объекта результата самостоятельно. Возвращая ошибку, необходимо выполнить очистку данных, так как событие PGEVT_RESULTDESTROY для этого объекта не поступит.

PGEVT_RESULTCOPY

Событие копирования объекта результата происходит при выполнении функции PQcopyResult. Это событие происходит только после того, как копирование будет завершено. Только те обработчики событий, которые успешно обработали событие PGEVT_RESULTCREATE или PGEVT_RESULTCOPY для исходного объекта, получат событие PGEVT_RESULTCOPY.

```
typedef struct
{
    const PGresult *src;
    PGresult *dest;
} PGEventResultCopy;
```

При поступлении события PGEVT_RESULTCOPY указатель *evtInfo* следует привести к PGEventResultCopy *. Поле *src* указывает на объект результата, который копируется, а *dest* — на целевой объект. Это событие может применяться для реализации внутреннего копирования instanceData, так как сама функция PQcopyResult не может это сделать. В случае сбоя обработчика вся операция копирования прерывается и объект результата в *dest* очищается. Возвращая ошибку, необходимо выполнить очистку данных целевого объекта, так как событие PGEVT_RESULTDESTROY для него не поступит.

PGEVT_RESULTDESTROY

Событие уничтожения объекта результата происходит при выполнении PQclear. Обработчик этого события отвечает за корректную очистку своих данных событий, так как libpq не может управлять его памятью. Невыполнение очистки должным образом приведёт к утечкам памяти.

```
typedef struct
{
    PGresult *result;
} PGEventResultDestroy;
```

При поступлении события PGEVT_RESULTDESTROY указатель *evtInfo* следует привести к PGEventResultDestroy *. Это событие происходит перед тем, как PQclear производит всю остальную очистку. Значение, возвращаемое обработчиком событий, игнорируется, так как из PQclear никак нельзя сообщить об ошибке. Кроме того, ошибка в обработчике событий не должна прерывать процесс очистки ставшей ненужной памяти.

31.13.2. Процедура обработки событий

PGEventProc

PGEventProc — это определение типа для указателя на обработчик событий, то есть функцию обратного вызова, получающую события от libpq. Обработчик событий должен иметь такую сигнатуру:

```
int eventproc(PGEventId evtId, void *evtInfo, void *passThrough)
```

Параметр *evtId* говорит, какое событие PGEVT произошло. Указатель *evtInfo* должен приводиться к типу определённой структуры для получения дополнительной информации о событии. В параметре *passThrough* передаётся сквозной указатель, поступивший в PQregisterEventProc при регистрации обработчика события. Эта функция должна вернуть ненулевое значение в случае успеха или ноль в противном случае.

Обработчик определённого события может быть зарегистрирован в любом PGconn только раз. Это связано с тем, что адрес обработчика используется как ключ для выбора связанных данных экземпляра.

> **Внимание**
>
> В Windows функции могут иметь два разных адреса: один, видимый снаружи DLL, и второй, видимый внутри DLL. Учитывая это, надо позаботиться о том, чтобы только один из адресов использовался с функциями обработки событий libpq, иначе возникнет путаница. Самый простой способ написать код, который будет работать — всегда помечать обработчик событий как `static`. Если адрес обработчика нужно получить вне его исходного файла, экспортируйте отдельную функцию, которая будет возвращать этот адрес.

31.13.3. Функции поддержки событий

`PQregisterEventProc`

Регистрирует обработчик событий в libpq.

```
int PQregisterEventProc(PGconn *conn, PGEventProc proc,
                        const char *name, void *passThrough);
```

Обработчик событий должен быть зарегистрирован один раз для каждого соединения `PGconn`, события которого представляют интерес. Число обработчиков событий, которые можно зарегистрировать для соединения, не ограничивается ничем, кроме объёма памяти. Эта функция возвращает ненулевое значение в случае успеха или ноль в противном случае.

Процедура, переданная в аргументе *proc*, будет вызываться, когда произойдёт событие libpq. Её адрес в памяти также применяется для поиска данных `instanceData`. Аргумент *name* используется при упоминании обработчика событий в сообщениях об ошибках. Это значение не может быть равно `NULL` или указывать на строку нулевой длины. Эта строка имени копируется в `PGconn`, так что переданная строка может быть временной. Сквозной указатель (*passThrough*) будет передаваться обработчику *proc* при каждом вызове события. Этот аргумент может равняться `NULL`.

`PQsetInstanceData`

Устанавливает для подключения *conn* указатель `instanceData` для обработчика *proc* равным *data*. Эта функция возвращает ненулевое значение в случае успеха или ноль в противном случае. (Ошибка возможна, только если обработчик *proc* не был корректно зарегистрирован для соединения *conn*.)

```
int PQsetInstanceData(PGconn *conn, PGEventProc proc, void *data);
```

`PQinstanceData`

Возвращает для соединения *conn* указатель на `instanceData`, связанный с обработчиком *proc*, либо `NULL`, если такого обработчика нет.

```
void *PQinstanceData(const PGconn *conn, PGEventProc proc);
```

`PQresultSetInstanceData`

Устанавливает для объекта результата (*res*) указатель `instanceData` для обработчика *proc* равным *data*. Эта функция возвращает ненулевое значение в случае успеха или ноль в противном случае. (Ошибка возможна, только если обработчик *proc* не был корректно зарегистрирован для объекта результата.)

```
int PQresultSetInstanceData(PGresult *res, PGEventProc proc, void *data);
```

`PQresultInstanceData`

Возвращает для объекта результата (*res*) указатель на `instanceData`, связанный с обработчиком *proc*, либо `NULL`, если такого обработчика нет.

```
void *PQresultInstanceData(const PGresult *res, PGEventProc proc);
```

31.13.4. Пример обработки событий

Ниже показан схематичный пример управления внутренними данными, связанными с подключениями и результатами libpq.

```
/* required header for libpq events (note: includes libpq-fe.h) */
#include <libpq-events.h>

/* The instanceData */
typedef struct
{
    int n;
    char *str;
} mydata;

/* PGEventProc */
static int myEventProc(PGEventId evtId, void *evtInfo, void *passThrough);

int
main(void)
{
    mydata *data;
    PGresult *res;
    PGconn *conn =
        PQconnectdb("dbname=postgres options=-csearch_path=");

    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        PQfinish(conn);
        return 1;
    }

    /* called once on any connection that should receive events.
     * Sends a PGEVT_REGISTER to myEventProc.
     */
    if (!PQregisterEventProc(conn, myEventProc, "mydata_proc", NULL))
    {
        fprintf(stderr, "Cannot register PGEventProc\n");
        PQfinish(conn);
        return 1;
    }

    /* conn instanceData is available */
    data = PQinstanceData(conn, myEventProc);

    /* Sends a PGEVT_RESULTCREATE to myEventProc */
    res = PQexec(conn, "SELECT 1 + 1");

    /* result instanceData is available */
    data = PQresultInstanceData(res, myEventProc);

    /* If PG_COPYRES_EVENTS is used, sends a PGEVT_RESULTCOPY to myEventProc */
    res_copy = PQcopyResult(res, PG_COPYRES_TUPLES | PG_COPYRES_EVENTS);

    /* result instanceData is available if PG_COPYRES_EVENTS was
```

```
     * used during the PQcopyResult call.
     */
    data = PQresultInstanceData(res_copy, myEventProc);

    /* Both clears send a PGEVT_RESULTDESTROY to myEventProc */
    PQclear(res);
    PQclear(res_copy);

    /* Sends a PGEVT_CONNDESTROY to myEventProc */
    PQfinish(conn);

    return 0;
}

static int
myEventProc(PGEventId evtId, void *evtInfo, void *passThrough)
{
    switch (evtId)
    {
        case PGEVT_REGISTER:
        {
            PGEventRegister *e = (PGEventRegister *)evtInfo;
            mydata *data = get_mydata(e->conn);

            /* associate app specific data with connection */
            PQsetInstanceData(e->conn, myEventProc, data);
            break;
        }

        case PGEVT_CONNRESET:
        {
            PGEventConnReset *e = (PGEventConnReset *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            if (data)
              memset(data, 0, sizeof(mydata));
            break;
        }

        case PGEVT_CONNDESTROY:
        {
            PGEventConnDestroy *e = (PGEventConnDestroy *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            /* free instance data because the conn is being destroyed */
            if (data)
              free_mydata(data);
            break;
        }

        case PGEVT_RESULTCREATE:
        {
            PGEventResultCreate *e = (PGEventResultCreate *)evtInfo;
            mydata *conn_data = PQinstanceData(e->conn, myEventProc);
            mydata *res_data = dup_mydata(conn_data);

            /* associate app specific data with result (copy it from conn) */
            PQsetResultInstanceData(e->result, myEventProc, res_data);
```

```
            break;
        }

        case PGEVT_RESULTCOPY:
        {
            PGEventResultCopy *e = (PGEventResultCopy *)evtInfo;
            mydata *src_data = PQresultInstanceData(e->src, myEventProc);
            mydata *dest_data = dup_mydata(src_data);

            /* associate app specific data with result (copy it from a result) */
            PQsetResultInstanceData(e->dest, myEventProc, dest_data);
            break;
        }

        case PGEVT_RESULTDESTROY:
        {
            PGEventResultDestroy *e = (PGEventResultDestroy *)evtInfo;
            mydata *data = PQresultInstanceData(e->result, myEventProc);

            /* free instance data because the result is being destroyed */
            if (data)
              free_mydata(data);
            break;
        }

        /* unknown event ID, just return TRUE. */
        default:
            break;
    }

    return TRUE; /* event processing succeeded */
}
```

31.14. Переменные окружения

Воспользовавшись следующими переменными окружения, можно задать значения параметров соединения по умолчанию, которые будут использоваться функциями `PQconnectdb`, `PQsetdbLogin` и `PQsetdb`, если никакое значение не будет задано вызывающим кодом. В частности, используя их, можно обойтись без жёсткого задания параметров соединения в простых клиентских приложениях.

- `PGHOST` действует так же, как параметр соединения host.
- `PGHOSTADDR` действует так же, как параметр соединения hostaddr. Эту переменную можно задать вместо или вместе с `PGHOST` для предотвращения поиска адреса в DNS.
- `PGPORT` действует так же, как параметр соединения port.
- `PGDATABASE` действует так же, как параметр соединения dbname.
- `PGUSER` действует так же, как параметр соединения user.
- `PGPASSWORD` действует так же, как параметр соединения password. Использовать эту переменную окружения не рекомендуется по соображениям безопасности, так как в некоторых операционных системах непривилегированные пользователи могут видеть переменные окружения процессов в выводе ps; вместо этого лучше использовать файл паролей (см. Раздел 31.15).
- `PGPASSFILE` действует так же, как параметр соединения passfile.
- `PGSERVICE` действует так же, как параметр соединения service.

- PGSERVICEFILE задаёт имя личного файла пользователя с параметрами подключения к службам (см. Раздел 31.16). По умолчанию применяется имя файла ~/.pg_service.conf (или %APPDATA%\postgresql\.pg_service.conf в Microsoft Windows).
- PGOPTIONS действует так же, как параметр соединения options.
- PGAPPNAME действует так же, как параметр соединения application_name.
- PGSSLMODE действует так же, как параметр соединения sslmode.
- PGREQUIRESSL действует так же, как параметр соединения requiressl. Эта переменная окружения утратила актуальность с появлением переменной PGSSLMODE; если установить обе переменные, значение данной не возымеет эффекта.
- PGSSLCOMPRESSION действует так же, как параметр соединения sslcompression.
- PGSSLCERT действует так же, как параметр соединения sslcert.
- PGSSLKEY действует так же, как параметр соединения sslkey.
- PGSSLROOTCERT действует так же, как параметр соединения sslrootcert.
- PGSSLCRL действует так же, как параметр соединения sslcrl.
- PGREQUIREPEER действует так же, как параметр соединения requirepeer.
- PGKRBSRVNAME действует так же, как параметр соединения krbsrvname.
- PGGSSLIB действует так же, как параметр соединения gsslib.
- PGCONNECT_TIMEOUT действует так же, как параметр соединения connect_timeout.
- PGCLIENTENCODING действует так же, как параметр соединения client_encoding.
- PGTARGETSESSIONATTRS действует так же, как параметр соединения target_session_attrs.

Следующие переменные окружения позволяют задать поведение по умолчанию для каждого отдельного сеанса Postgres Pro. (См. также описание команд ALTER ROLE и ALTER DATABASE, позволяющих установить поведение по умолчанию для отдельного пользователя или отдельной базы.)

- PGDATESTYLE устанавливает стиль представления даты/времени по умолчанию. (Равносильно SET datestyle TO ....)
- PGTZ устанавливает часовой пояс по умолчанию. (Равносильно SET timezone TO ....)
- PGGEQO устанавливает режим по умолчанию для генетического оптимизатора запросов. (Равносильно SET geqo TO ....)

Информацию о корректных значениях этих переменных окружения можно найти в описании SQL-команды SET.

Следующие переменные среды определяют внутреннее поведение libpq; они переопределяют встроенные значения.

- PGSYSCONFDIR задаёт каталог, в котором содержится файл pg_service.conf, а в будущем он может содержать и другие общесистемные файлы конфигурации.
- PGLOCALEDIR задаёт каталог, содержащий файлы locale, предназначенные для перевода сообщений.

31.15. Файл паролей

Файл .pgpass в домашнем каталоге пользователя может содержать пароли, которые будут использоваться, если для подключения требуется пароль (и пароль не задаётся другим способом). В Microsoft Windows этот файл называется %APPDATA%\postgresql\pgpass.conf (где %APPDATA% обозначает каталог с данными приложений в профиле пользователя). Имя используемого файла паролей также можно задать в параметре подключения passfile или в переменной окружения PGPASSFILE.

Этот файл должен содержать строки следующего формата:

сервер:*порт*:*база_данных*:*имя_пользователя*:*пароль*

(Вы можете вставить в этот файл комментарий-памятку, скопировав показанную строку в него и добавив в начало #.) Первые четыре поля могут содержать строковые значения, либо знак *, соответствующий всему. Применяться будет пароль, указанный в первой из строк, значения полей в которой соответствуют текущему соединению. (Поэтому, если вы используете звёздочки, поместите более конкретные записи первыми.) Если запись должна содержать символ : или \, добавьте перед ним \. Поле с именем узла сопоставляется с параметром подключения host (если он указан) или с параметром hostaddr (если указан он); в случае отсутствия обоих параметров подразумевается имя localhost. Имя узла localhost также подразумевается, когда соединение устанавливается через Unix-сокет и параметр host соответствует установленному в libpq каталогу сокетов по умолчанию. На ведомом сервере имя базы данных replication соответствует подключениям к ведущему серверу, которые применяются для потоковой репликации. Поле *база_данных* имеет ограниченную ценность, так как пользователи используют один пароль для всех баз данных в кластере.

В системах Unix разрешения для файла паролей должны запрещать любой доступ к нему всем и группе; этого можно добиться командой chmod 0600 ~/.pgpass. Если разрешения будут менее строгими, этот файл будет игнорироваться. В Microsoft Windows предполагается, что файл хранится в безопасном месте, и никакие дополнительные проверки не производятся.

31.16. Файл соединений служб

Файл соединений служб позволяет связать параметры соединений libpq с одним именем службы. Затем это имя службы можно задать в строке подключения libpq и будут применены все связанные с ним параметры. Это позволяет модифицировать параметры соединений, обходясь без перекомпиляции приложения, использующего libpq. Имя службы можно также задать в переменной окружения PGSERVICE.

Имена служб могут быть определены либо в служебном файле каждого пользователя, либо в общесистемном файле. Если одно и то же имя службы существует как в пользовательском, так и в системном файле, пользовательский файл имеет приоритет. По умолчанию служебный файл пользователя называется ~/.pg_service.conf. В Microsoft Windows этот файл называется %APPDATA%\postgresql\pg_service.conf (где %APPDATA% обозначает каталог данных приложений (Application Data) в профиле пользователя). Имя файла также можно задать в переменной окружения PGSERVICEFILE. Общесистемный файл называется pg_service.conf. По умолчанию он ищется в каталоге etc установки PostgreSQL (используйте pg_config --sysconfdir, чтобы точно установить этот каталог). Можно указать другой каталог, но не другое имя файла, задав переменную окружения PGSYSCONFDIR.

В любом служебном файле используется формат «INI-файлов», в котором имя раздела задаёт имя службы, а параметры внутри — параметры соединения; их список приведён в Подразделе 31.1.2. Например:

```
# комментарий
[mydb]
host=somehost
port=5433
user=admin
```

Пример такого файла можно найти в каталоге установки PostgreSQL в share/pg_service.conf.sample.

Параметры подключения, полученные из служебного файла, объединяются с параметрами, полученными из других источников. Значение в этом файле переопределяет соответствующую переменную окружения и, в свою очередь, может быть изменён значением, указанным непосредственно в строке подключения. Например, используя указанный выше служебный файл, строка подключения service=mydb port=5434 будет использовать сервер somehost, порт 5434, пользователя admin и другие параметры, установленные переменными окружения или встроенными значениями по умолчанию.

31.17. Получение параметров соединения через LDAP

Если библиотека libpq была собрана с поддержка LDAP (`configure` передавался ключ `--with-ldap`), такие параметры соединения, как `host` и `dbname`, можно получить через LDAP с центрального сервера. Преимущество такого подхода в том, что при изменении параметров подключения к базе данных свойства соединения не придётся изменять на всех клиентских компьютерах.

Для получения параметров соединений через LDAP используется файл соединений служб `pg_service.conf` (см. Раздел 31.16). Строка в `pg_service.conf`, начинающаяся с указания протокола `ldap://`, будет воспринята как URL в LDAP и выполнится как запрос к LDAP. Результатом запроса должен быть список пар `keyword = value`, которые и будут задавать параметры соединений. Заданный URL должен соответствовать RFC 1959 и иметь следующий вид:

`ldap://[`*`имя_сервера`*`[:`*`порт`*`]]/`*`база_поиска`*`?`*`атрибут`*`?`*`область_поиска`*`?`*`фильтр`*

; по умолчанию *`имя_сервера`* — `localhost`, а *`порт`* — 389.

Обработка `pg_service.conf` прекращается после удачного поиска в LDAP, но если с сервером LDAP связаться не удаётся, обрабатываются следующие строки этого файла. Так сделано для того, чтобы можно было реализовать запасные варианты, добавив дополнительные строки с URL LDAP, указывающими на другие серверы LDAP, или классические пары `keyword = value`, либо используя параметры соединений по умолчанию. Если же вы хотите получить ошибку в этой ситуации, добавьте после строки с URL-адресом LDAP синтаксически некорректную строку.

Простую запись LDAP, созданную из такого файла LDIF

```
version:1
dn:cn=mydatabase,dc=mycompany,dc=com
changetype:add
objectclass:top
objectclass:device
cn:mydatabase
description:host=dbserver.mycompany.com
description:port=5439
description:dbname=mydb
description:user=mydb_user
description:sslmode=require
```

можно запросить из каталога LDAP, указав следующий URL:

```
ldap://ldap.mycompany.com/dc=mycompany,dc=com?description?one?(cn=mydatabase)
```

Также возможно сочетать обычные записи в файле служб с поиском в LDAP. Полный пример описания службы в `pg_service.conf` может быть таким:

```
# в LDAP хранится только имя и порт сервера; имя базы и пользователя нужно задать явно
[customerdb]
dbname=customer
user=appuser
ldap://ldap.acme.com/cn=dbserver,cn=hosts?pgconnectinfo?base?(objectclass=*)
```

31.18. Поддержка SSL

Postgres Pro реализует собственную поддержку SSL-подключений для шифрования клиент-серверного взаимодействия в качестве меры безопасности. Подробнее функциональность SSL на стороне сервера описывается в Разделе 17.9.

Библиотека libpq читает системный файл конфигурации OpenSSL. По умолчанию этот файл называется `openssl.cnf` и находится в каталоге, который сообщает команда `openssl version -d`. Если требуется указать другое расположение файла конфигурации, его можно задать в переменной окружения `OPENSSL_CONF`.

31.18.1. Проверка сертификатов сервера на стороне клиента

По умолчанию Postgres Pro не выполняет никакие проверки сертификата сервера. Это означает, что клиента можно ввести в заблуждение, подменив сервер (например, изменив запись в DNS или заняв его IP-адрес). Чтобы защититься от такой подмены, клиент должен иметь возможность проверять подлинность сервера по цепочке доверия. Для создания цепочки доверия нужно установить корневой (самоподписанный) сертификат центра сертификации (ЦС) на одном компьютере, а конечный сертификат, *подписанный* корневым, на другом. В цепочке может использоваться и «промежуточный» сертификат, который подписывается корневым сертификатом и подписывает подчинённые сертификаты.

Чтобы клиент мог проверить подлинность сервера, установите на клиенте корневой сертификат, а на сервере установите конечный сертификат, подписанный этим корневым. Чтобы сервер мог проверить подлинность клиента, установите на сервере корневой сертификат, а на клиенте конечный сертификат, подписанный данным корневым. Конечный сертификат также может связываться с корневым через один или несколько промежуточных сертификатов (они обычно хранятся вместе с конечным сертификатом).

Когда цепочка доверия присутствует, клиент может проверить конечный сертификат, переданный сервером, двумя способами. Если параметр `sslmode` имеет значение `verify-ca`, libpq будет проверять подлинность сервера, проверяя цепочку доверия до корневого сертификата, установленного на клиенте. Если в `sslmode` выбран режим `verify-full`, libpq будет *также* проверять соответствие имени узла сервера имени, записанному в сертификате. SSL-подключение не будет установлено, если проверить сертификат сервера не удастся. Режим `verify-full` рекомендуется для большинства окружений, где важна безопасность.

В режиме `verify-full` имя компьютера сверяется с атрибутом (или атрибутами) Subject Alternative Name (Альтернативное имя субъекта) в сертификате или с атрибутом Common Name (Общее имя), если в сертификате отсутствует атрибут Subject Alternative Name типа `dNSName`. Если атрибут имени сертификата начинается со звёздочки (`*`), звёздочка воспринимается как подстановочный знак и ей будут соответствовать все символы, *кроме* точки (`.`). Это означает, что такой сертификат не будет соответствовать поддоменам. Если подключение устанавливается по IP-адресу, а не по имени компьютера, проверяться будет IP-адрес (без поиска в DNS).

Чтобы настроить проверку сертификата сервера, необходимо поместить один или несколько корневых сертификатов в файл `~/.postgresql/root.crt` в домашнем каталоге пользователя. (В Microsoft Windows этот файл называется `%APPDATA%\postgresql\root.crt`.) Также следует добавлять в этот файл промежуточные сертификаты, если они нужны для связывания цепочки сертификатов, переданных сервером, с корневыми сертификатами, установленными на клиенте.

Если существует файл `~/.postgresql/root.crl` (или `%APPDATA%\postgresql\root.crl` в Microsoft Windows), при проверке также учитывается содержащийся в нём список отозванных сертификатов (CRL, Certificate Revocation List).

Размещение файла корневых сертификатов и CRL можно поменять, задав параметры соединения `sslrootcert` и `sslcrl` или переменные окружения `PGSSLROOTCERT` и `PGSSLCRL`, соответственно.

> **Примечание**
>
> Для обратной совместимости с предыдущими версиями Postgres Pro, при наличии файла с сертификатами корневых ЦС поведение режима `sslmode=require` не отличается от режима `verify-ca`, то есть сертификат сервера будет проверяться по сертификату ЦС. Полагаться на это поведение не рекомендуется — приложения, которым нужно проверять сертификат, должны всегда выбирать режим `verify-ca` или `verify-full`.

31.18.2. Клиентские сертификаты

Если сервер попытается проверить подлинность клиента, запрашивая конечный сертификат клиента, libpq будет передавать сертификаты, сохранённые в файле `~/.postgresql/`

postgresql.crt в домашнем каталоге пользователя. Эти сертификаты должны связываться по цепочке с корневым сертификатом, которому доверяет сервер. Также должен присутствовать соответствующий закрытый ключ ~/.postgresql/postgresql.key. В Microsoft Windows эти файлы называются %APPDATA%\postgresql\postgresql.crt и %APPDATA%\postgresql\postgresql.key. Размещение файлов сертификатов и закрытого ключа можно переопределить с помощью параметров подключения sslcert и sslkey либо переменных окружения PGSSLCERT и PGSSLKEY.

В системах Unix разрешения для файла закрытого ключа должны запрещать любой доступ к нему всем и группе; этого можно добиться командой chmod 0600 ~/.postgresql/postgresql.key. Возможен и другой вариант, когда этим файлом владеет root, а группа имеет доступ на чтение (то есть, маска разрешений 0640). Данный вариант полезен, когда файлами сертификатов и ключей управляет сама операционная система. В этом случае пользователь, запускающий libpq, должен быть членом группы, имеющей доступ к указанным файлам сертификата и ключа. (В Microsoft Windows каталог %APPDATA%\postgresql считается защищённым, поэтому разрешения для файлов не проверяются.)

Первым сертификатом в postgresql.crt должен быть сертификат клиента, так как он должен соответствовать закрытому ключу клиента. Дополнительно в этот файл могут быть добавлены «промежуточные» сертификаты — это позволит избежать хранения всех промежуточных сертификатов в файле root.crt на сервере.

За инструкциями по созданию сертификатов обратитесь к Подразделу 17.9.3.

31.18.3. Защита, обеспечиваемая в различных режимах

Разные значения параметра sslmode обеспечивают разные уровни защиты. SSL позволяет защититься от следующих типов атак:

Прослушивание

Если третья сторона может прослушивать сетевой трафик между клиентом и сервером, она может получить как информацию соединения (включая имя пользователя и пароль), так и передаваемые данные. Чтобы защититься от этого, SSL шифрует трафик.

Посредник (MITM)

Если третья сторона может модифицировать данные, передаваемые между клиентом и сервером, она может представиться сервером и, таким образом, сможет видеть и модифицировать данные, *даже если они зашифрованы*. Третья сторона затем может воспроизводить характеристики соединения и данные для подлинного сервера, что сделает невозможным обнаружение этой атаки. Векторами такой атаки может быть «отравление» DNS и подмена адресов, в результате чего клиент будет обращаться не к тому серверу, к которому нужно. Также есть несколько других вариантов реализации этой атаки. Для защиты в SSL применяется проверка сертификатов, в результате которой сервер доказывает свою подлинность клиенту.

Олицетворение

Если третья сторона может представляться авторизованным клиентом, она может просто обращаться к данным, к которым не должна иметь доступа. Обычно это происходит вследствие небезопасного управления паролями. В SSL для предотвращения этой угрозы используются клиентские сертификаты, гарантирующие, что к серверу могут обращаться только владельцы действительных сертификатов.

Чтобы соединение было гарантированно безопасным, SSL должен быть настроен *на клиенте и на сервере*, прежде чем будет установлено соединение. Если он настроен только на сервере, клиент может начать передавать важную информацию (например, пароли), до того как поймёт, что сервер требует высокого уровня безопасности. В libpq для установления безопасных соединений нужно задать для параметра sslmode значение verify-full или verify-ca и предоставить системе корневой сертификат для проверки. В качестве аналогии можно привести использование адреса с https для безопасного просмотра веб-содержимого.

Когда подлинность сервера подтверждена, клиент может передавать конфиденциальные данные. Это значит, что до этого момента клиенту не нужно знать, применяются ли сертификаты для аутентификации, так что настройка использования сертификатов только на стороне сервера не угрожает безопасности.

Все варианты использования SSL подразумевают издержки шифрования и обмена ключами, что порождает необходимость выбора между производительностью и безопасностью. В Таблице 31.1 описываются риски, от которых защищают различные варианты `sslmode`, и приводятся утверждения относительно защиты и издержек.

Таблица 31.1. Описания режимов SSL

`sslmode`	**Защита от прослушивания**	**Защита от MITM**	**Утверждение**
`disable`	Нет	Нет	Мне не важна безопасность и я не приемлю издержки, связанные с шифрованием.
`allow`	Возможно	Нет	Мне не важна безопасность, но я приемлю издержки, связанные с шифрованием, если на этом настаивает сервер.
`prefer`	Возможно	Нет	Мне не важна безопасность, но я предпочитаю шифрование (и приемлю связанные издержки), если это поддерживает сервер.
`require`	Да	Нет	Я хочу, чтобы мои данные шифровались, и я приемлю сопутствующие издержки. Я доверяю сети в том, что она обеспечивает подключение к нужному серверу.
`verify-ca`	Да	Зависит от политики ЦС	Я хочу, чтобы мои данные шифровались, и я приемлю сопутствующие издержки. Мне нужна уверенность в том, что я подключаюсь к доверенному серверу.
`verify-full`	Да	Да	Я хочу, чтобы мои данные шифровались, и я приемлю сопутствующие издержки. Мне нужна уверенность в том, что я подключаюсь к

sslmode	**Защита от прослушивания**	**Защита от MITM**	**Утверждение**
			доверенному серверу и это именно указанный мной сервер.

Различие вариантов `verify-ca` и `verify-full` зависит от характера корневого ЦС. Если используется публичный ЦС, режим `verify-ca` допускает подключение к серверу с сертификатом, который получил *кто угодно* в этом ЦС. В такой ситуации нужно всегда использовать режим `verify-full`. Если же используется локальный ЦС или даже самоподписанный сертификат, режим `verify-ca` обычно обеспечивает достаточную защиту.

По умолчанию параметр `sslmode` имеет значение `prefer`. Как показано в таблице, оно неэффективно с точки зрения безопасности и может только привносить дополнительные издержки. Оно выбрано по умолчанию исключительно для обратной совместимости и не рекомендуется для защищённых окружений.

31.18.4. Файлы, используемые клиентом SSL

В Таблице 31.2 перечислены файлы, имеющие отношение к настройке SSL на стороне клиента.

Таблица 31.2. Файлы, используемые клиентом SSL/libpq

Файл	**Содержимое**	**Назначение**
`~/.postgresql/postgresql.crt`	сертификат клиента	передаётся серверу
`~/.postgresql/postgresql.key`	закрытый ключ клиента	подтверждает клиентский сертификат, передаваемый владельцем; не гарантирует, что владелец сертификата заслуживает доверия
`~/.postgresql/root.crt`	сертификаты доверенных ЦС	позволяет проверить, что сертификат сервера подписан доверенным центром сертификации
`~/.postgresql/root.crl`	сертификаты, отозванные центрами сертификации	сертификат сервера должен отсутствовать в этом списке

31.18.5. Инициализация библиотеки SSL

Если ваше приложение инициализирует библиотеку `libssl` и/или `libcrypto`, и libpq собрана с поддержкой SSL, вы должны вызвать `PQinitOpenSSL`, чтобы сообщить libpq, что библиотека `libssl` и/или `libcrypto` уже инициализированы вашим приложением, чтобы libpq не пыталась ещё раз инициализировать их.

`PQinitOpenSSL`

Позволяет приложениям выбрать, какие библиотеки безопасности нужно инициализировать.

```
void PQinitOpenSSL(int do_ssl, int do_crypto);
```

Когда параметр *`do_ssl`* отличен от нуля, libpq будет инициализировать библиотеку OpenSSL перед первым подключением к базе данных. Когда параметр *`do_crypto`* не равен нулю, будет инициализироваться библиотека `libcrypto`. По умолчанию (если функция `PQinitOpenSSL` не вызывается) инициализируются обе библиотеки. Если поддержка SSL не была скомпилирована, эта функция присутствует, но ничего не делает.

Если ваше приложение использует и инициализирует библиотеку OpenSSL или её нижележащую библиотеку `libcrypto`, вы *должны* вызвать эту функцию, передав нули в соответствующих параметрах, перед первым подключением к базе данных. Собственно инициализацию также важно произвести перед установлением подключения.

`PQinitSSL`

Позволяет приложениям выбрать, какие библиотеки безопасности нужно инициализировать.

```
void PQinitSSL(int do_ssl);
```

Эта функция равнозначна вызову `PQinitOpenSSL(do_ssl, do_ssl)`. Приложениям достаточно инициализировать или не инициализировать обе библиотеки OpenSSL и `libcrypto` одновременно.

Функция `PQinitSSL` существует со времён PostgreSQL 8.0, тогда как `PQinitOpenSSL` появилась в PostgreSQL 8.4, так что `PQinitSSL` может быть предпочтительней для приложений, которым нужно работать с более старыми версиями libpq.

31.19. Поведение в многопоточных программах

Библиотека libpq по умолчанию поддерживает повторные вызовы и многопоточность. Для соответствующего варианта сборки вашего приложения вам может понадобиться передать компилятору специальные параметры командной строки. Чтобы узнать, как собрать многопоточное приложение, обратитесь к документации вашей системы или поищите в файле `src/Makefile.global` значения `PTHREAD_CFLAGS` и `PTHREAD_LIBS`. Эта функция позволяет узнать, поддерживает ли libpq многопоточность:

`PQisthreadsafe`

Возвращает состояние потокобезопасности в библиотеке libpq.

```
int PQisthreadsafe();
```

Возвращает 1, если библиотека libpq потокобезопасная, или 0 в противном случае.

Реализация многопоточности не лишена ограничений: два потока не должны пытаться одновременно работать с одним объектом `PGconn`. В частности, не допускается параллельное выполнение команд из разных потоков через один объект соединения. (Если вам нужно выполнять команды одновременно, используйте несколько соединений.)

Объекты `PGresult` после создания обычно доступны только для чтения, и поэтому их можно свободно передавать между потоками. Однако если вы используете какую-либо из функций, изменяющих `PGresult`, описанных в Разделе 31.11 или Разделе 31.13, вы должны также избегать одновременных обращений к одному объекту `PGresult`.

Устаревшие функции `PQrequestCancel` и `PQoidStatus` не являются потокобезопасными и не должны применяться в многопоточных программах. Вместо `PQrequestCancel` можно использовать `PQcancel`, а вместо `PQoidStatus` — `PQoidValue`.

Если вы применяете Kerberos в своём приложении (помимо возможного использования внутри libpq), вы должны обеспечить блокировку вокруг вызовов Kerberos, так как функции Kerberos не являются потокобезопасными. Обратите внимание на функцию `PQregisterThreadLock` в исходном коде libpq, позволяющую организовать совместные блокировки между libpq и вашим приложением.

31.20. Сборка программ с libpq

Чтобы собрать (то есть, скомпилировать и скомпоновать) программу, использующую libpq, вы должны проделать следующие действия:

- Включите заголовочный файл `libpq-fe.h`:

  ```
  #include <libpq-fe.h>
  ```

  Если вы не сделаете этого, обычно вас ждут примерно такие сообщения об ошибках от компилятора:

  ```
  foo.c: In function `main':
  foo.c:34: `PGconn' undeclared (first use in this function)
  ```

```
foo.c:35: `PGresult' undeclared (first use in this function)
foo.c:54: `CONNECTION_BAD' undeclared (first use in this function)
foo.c:68: `PGRES_COMMAND_OK' undeclared (first use in this function)
foo.c:95: `PGRES_TUPLES_OK' undeclared (first use in this function)
```

- Сообщите вашему компилятору каталог, в котором установлены заголовочные файлы Postgres Pro, передав ему параметр `-I`*`каталог`*. (В некоторых случаях компилятор сам может обращаться к нужному каталогу, так что этот параметр можно опустить.) Например, ваша команда компиляции может быть такой:

  ```
  cc -c -I/usr/local/pgsql/include testprog.c
  ```

  Если вы используете скрипты сборки Makefile, добавьте этот параметр в переменную `CPPFLAGS`:

  ```
  CPPFLAGS += -I/usr/local/pgsql/include
  ```

  Если существует возможность, что вашу программу будут компилировать другие пользователи, то путь к каталогу не следует жёстко задавать таким образом. Вместо этого вы можете воспользоваться утилитой `pg_config`и узнать, где в локальной системе находятся заголовочные файлы, следующим образом:

  ```
  $ pg_config --includedir
  /usr/local/include
  ```

  Если у вас установлена программа `pkg-config`, вместо этого вы можете выполнить:

  ```
  $ pkg-config --cflags libpq
  -I/usr/local/include
  ```

  Заметьте, что при этом перед путём сразу будет добавлен ключ `-I`.

  Если требуемый параметр не будет передан компилятору, вы получите примерно такое сообщение об ошибке:

  ```
  testlibpq.c:8:22: libpq-fe.h: No such file or directory
  ```

- При компоновке окончательной программы добавьте параметр `-lpq`, чтобы была подключена библиотека libpq, а также параметр `-L`*`каталог`*, указывающий на каталог, в котором находится libpq. (Опять же, компилятор будет просматривать определённые каталоги по умолчанию.) Для максимальной переносимости указывайте ключ `-L` перед параметром `-lpq`. Например:

  ```
  cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -lpq
  ```

  Каталог с библиотекой можно узнать, так же используя `pg_config`:

  ```
  $ pg_config --libdir
  /usr/local/pgsql/lib
  ```

  Или с помощью той же программы `pkg-config`:

  ```
  $ pkg-config --libs libpq
  -L/usr/local/pgsql/lib -lpq
  ```

  Заметьте, что и в этом случае выводится полностью сформированный параметр, а не только путь.

  В случае проблем в этой области возможны примерно такие сообщения об ошибках:

  ```
  testlibpq.o: In function `main':
  testlibpq.o(.text+0x60): undefined reference to `PQsetdbLogin'
  testlibpq.o(.text+0x71): undefined reference to `PQstatus'
  testlibpq.o(.text+0xa4): undefined reference to `PQerrorMessage'
  ```

  Они означают, что вы забыли добавить параметр `-lpq`.

  ```
  /usr/bin/ld: cannot find -lpq
  ```

Такая ошибка означает, что вы забыли добавить ключ `-L` или не указали правильный каталог.

31.21. Примеры программ

Эти и другие примеры можно найти в каталоге `src/test/examples` в дистрибутиве исходного кода.

Пример 31.1. Первая программа, демонстрирующая использование libpq

```
/*
 * src/test/examples/testlibpq.c
 *
 *
 * testlibpq.c
 *
 *      Test the C version of libpq, the PostgreSQL frontend library.
 */
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    int         nFields;
    int         i,
                j;

    /*
     * If the user supplies a parameter on the command line, use it as the
     * conninfo string; otherwise default to setting dbname=postgres and using
     * environment variables or defaults for all other connection parameters.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* Make a connection to the database */
    conn = PQconnectdb(conninfo);

    /* Check to see that the backend connection was successfully made */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }
```

```
/* Set always-secure search path, so malicious users can't take control. */
res = PQexec(conn,
             "SELECT pg_catalog.set_config('search_path', '', false)");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/*
 * Should PQclear PGresult whenever it is no longer needed to avoid memory
 * leaks
 */
PQclear(res);

/*
 * Our test case here involves using a cursor, for which we must be inside
 * a transaction block.  We could do the whole thing with a single
 * PQexec() of "select * from pg_database", but that's too trivial to make
 * a good example.
 */

/* Start a transaction block */
res = PQexec(conn, "BEGIN");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "BEGIN command failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

/*
 * Fetch rows from pg_database, the system catalog of databases
 */
res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from pg_database");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "DECLARE CURSOR failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

res = PQexec(conn, "FETCH ALL in myportal");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "FETCH ALL failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/* first, print out the attribute names */
nFields = PQnfields(res);
for (i = 0; i < nFields; i++)
    printf("%-15s", PQfname(res, i));
```

```
    printf("\n\n");

    /* next, print out the rows */
    for (i = 0; i < PQntuples(res); i++)
    {
        for (j = 0; j < nFields; j++)
            printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }

    PQclear(res);

    /* close the portal ... we don't bother to check for errors ... */
    res = PQexec(conn, "CLOSE myportal");
    PQclear(res);

    /* end the transaction */
    res = PQexec(conn, "END");
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Пример 31.2. Вторая программа, демонстрирующая использование libpq

```
/*
 * src/test/examples/testlibpq2.c
 *
 *
 * testlibpq2.c
 *      Test of the asynchronous notification interface
 *
 * Start this program, then from psql in another window do
 *   NOTIFY TBL2;
 * Repeat four times to get this program to exit.
 *
 * Or, if you want to get fancy, try this:
 * populate a database with the following commands
 * (provided in src/test/examples/testlibpq2.sql):
 *
 *   CREATE SCHEMA TESTLIBPQ2;
 *   SET search_path = TESTLIBPQ2;
 *   CREATE TABLE TBL1 (i int4);
 *   CREATE TABLE TBL2 (i int4);
 *   CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *     (INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);
 *
 * Start this program, then from psql do this four times:
 *
 *   INSERT INTO TESTLIBPQ2.TBL1 VALUES (10);
 */

#ifdef WIN32
#include <windows.h>
```

```
#endif
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/types.h>
#ifdef HAVE_SYS_SELECT_H
#include <sys/select.h>
#endif

#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    PGnotify   *notify;
    int         nnotifies;

    /*
     * If the user supplies a parameter on the command line, use it as the
     * conninfo string; otherwise default to setting dbname=postgres and using
     * environment variables or defaults for all other connection parameters.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* Make a connection to the database */
    conn = PQconnectdb(conninfo);

    /* Check to see that the backend connection was successfully made */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* Set always-secure search path, so malicious users can't take control. */
    res = PQexec(conn,
                 "SELECT pg_catalog.set_config('search_path', '', false)");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
```

```
}

/*
 * Should PQclear PGresult whenever it is no longer needed to avoid memory
 * leaks
 */
PQclear(res);

/*
 * Issue LISTEN command to enable notifications from the rule's NOTIFY.
 */
res = PQexec(conn, "LISTEN TBL2");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "LISTEN command failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

/* Quit after four notifies are received. */
nnotifies = 0;
while (nnotifies < 4)
{
    /*
     * Sleep until something happens on the connection.  We use select(2)
     * to wait for input, but you could also use poll() or similar
     * facilities.
     */
    int         sock;
    fd_set      input_mask;

    sock = PQsocket(conn);

    if (sock < 0)
        break;              /* shouldn't happen */

    FD_ZERO(&input_mask);
    FD_SET(sock, &input_mask);

    if (select(sock + 1, &input_mask, NULL, NULL, NULL) < 0)
    {
        fprintf(stderr, "select() failed: %s\n", strerror(errno));
        exit_nicely(conn);
    }

    /* Now check for input */
    PQconsumeInput(conn);
    while ((notify = PQnotifies(conn)) != NULL)
    {
        fprintf(stderr,
                "ASYNC NOTIFY of '%s' received from backend PID %d\n",
                notify->relname, notify->be_pid);
        PQfreemem(notify);
        nnotifies++;
        PQconsumeInput(conn);
    }
}
```

```
    fprintf(stderr, "Done.\n");

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Пример 31.3. Третья программа, демонстрирующая использование libpq

```
/*
 * src/test/examples/testlibpq3.c
 *
 *
 * testlibpq3.c
 *      Test out-of-line parameters and binary I/O.
 *
 * Before running this, populate a database with the following commands
 * (provided in src/test/examples/testlibpq3.sql):
 *
 * CREATE SCHEMA testlibpq3;
 * SET search_path = testlibpq3;
 * CREATE TABLE test1 (i int4, t text, b bytea);
 * INSERT INTO test1 values (1, 'joe''s place', '\\000\\001\\002\\003\\004');
 * INSERT INTO test1 values (2, 'ho there', '\\004\\003\\002\\001\\000');
 *
 * The expected output is:
 *
 * tuple 0: got
 *  i = (4 bytes) 1
 *  t = (11 bytes) 'joe's place'
 *  b = (5 bytes) \000\001\002\003\004
 *
 * tuple 0: got
 *  i = (4 bytes) 2
 *  t = (8 bytes) 'ho there'
 *  b = (5 bytes) \004\003\002\001\000
 */

#ifdef WIN32
#include <windows.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <sys/types.h>
#include "libpq-fe.h"

/* for ntohl/htonl */
#include <netinet/in.h>
#include <arpa/inet.h>


static void
```

```
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

/*
 * This function prints a query result that is a binary-format fetch from
 * a table defined as in the comment above.  We split it out because the
 * main() function uses it twice.
 */
static void
show_binary_results(PGresult *res)
{
    int         i,
                j;
    int         i_fnum,
                t_fnum,
                b_fnum;

    /* Use PQfnumber to avoid assumptions about field order in result */
    i_fnum = PQfnumber(res, "i");
    t_fnum = PQfnumber(res, "t");
    b_fnum = PQfnumber(res, "b");

    for (i = 0; i < PQntuples(res); i++)
    {
        char       *iptr;
        char       *tptr;
        char       *bptr;
        int         blen;
        int         ival;

        /* Get the field values (we ignore possibility they are null!) */
        iptr = PQgetvalue(res, i, i_fnum);
        tptr = PQgetvalue(res, i, t_fnum);
        bptr = PQgetvalue(res, i, b_fnum);

        /*
         * The binary representation of INT4 is in network byte order, which
         * we'd better coerce to the local byte order.
         */
        ival = ntohl(*((uint32_t *) iptr));

        /*
         * The binary representation of TEXT is, well, text, and since libpq
         * was nice enough to append a zero byte to it, it'll work just fine
         * as a C string.
         *
         * The binary representation of BYTEA is a bunch of bytes, which could
         * include embedded nulls so we have to pay attention to field length.
         */
        blen = PQgetlength(res, i, b_fnum);

        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d\n",
               PQgetlength(res, i, i_fnum), ival);
        printf(" t = (%d bytes) '%s'\n",
```

```
                PQgetlength(res, i, t_fnum), tptr);
            printf(" b = (%d bytes) ", blen);
            for (j = 0; j < blen; j++)
                printf("\\%03o", bptr[j]);
            printf("\n\n");
        }
    }

    int
    main(int argc, char **argv)
    {
        const char *conninfo;
        PGconn     *conn;
        PGresult   *res;
        const char *paramValues[1];
        int         paramLengths[1];
        int         paramFormats[1];
        uint32_t    binaryIntVal;

        /*
         * If the user supplies a parameter on the command line, use it as the
         * conninfo string; otherwise default to setting dbname=postgres and using
         * environment variables or defaults for all other connection parameters.
         */
        if (argc > 1)
            conninfo = argv[1];
        else
            conninfo = "dbname = postgres";

        /* Make a connection to the database */
        conn = PQconnectdb(conninfo);

        /* Check to see that the backend connection was successfully made */
        if (PQstatus(conn) != CONNECTION_OK)
        {
            fprintf(stderr, "Connection to database failed: %s",
                    PQerrorMessage(conn));
            exit_nicely(conn);
        }

        /* Set always-secure search path, so malicious users can't take control. */
        res = PQexec(conn, "SET search_path = testlibpq3");
        if (PQresultStatus(res) != PGRES_COMMAND_OK)
        {
            fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
            PQclear(res);
            exit_nicely(conn);
        }
        PQclear(res);

        /*
         * The point of this program is to illustrate use of PQexecParams() with
         * out-of-line parameters, as well as binary transmission of data.
         *
         * This first example transmits the parameters as text, but receives the
         * results in binary format.  By using out-of-line parameters we can avoid
         * a lot of tedious mucking about with quoting and escaping, even though
         * the data is text.  Notice how we don't have to do anything special with
```

```
 * the quote mark in the parameter value.
 */

/* Here is our out-of-line parameter value */
paramValues[0] = "joe's place";

res = PQexecParams(conn,
                   "SELECT * FROM test1 WHERE t = $1",
                   1,       /* one param */
                   NULL,    /* let the backend deduce param type */
                   paramValues,
                   NULL,    /* don't need param lengths since text */
                   NULL,    /* default to all text params */
                   1);      /* ask for binary results */

if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

show_binary_results(res);

PQclear(res);

/*
 * In this second example we transmit an integer parameter in binary form,
 * and again retrieve the results in binary form.
 *
 * Although we tell PQexecParams we are letting the backend deduce
 * parameter type, we really force the decision by casting the parameter
 * symbol in the query text.  This is a good safety measure when sending
 * binary parameters.
 */

/* Convert integer value "2" to network byte order */
binaryIntVal = htonl((uint32_t) 2);

/* Set up parameter arrays for PQexecParams */
paramValues[0] = (char *) &binaryIntVal;
paramLengths[0] = sizeof(binaryIntVal);
paramFormats[0] = 1;        /* binary */

res = PQexecParams(conn,
                   "SELECT * FROM test1 WHERE i = $1::int4",
                   1,       /* one param */
                   NULL,    /* let the backend deduce param type */
                   paramValues,
                   paramLengths,
                   paramFormats,
                   1);      /* ask for binary results */

if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
```

```
    }

    show_binary_results(res);

    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Глава 32. Большие объекты

В Postgres Pro имеется механизм для работы с *большими объектами*, предоставляющий доступ в потоковом режиме к пользовательским данным, сохранённым в специальной структуре больших объектов. Потоковый доступ удобен, когда нужно обрабатывать данные, объём которых слишком велик, чтобы оперировать ими как единым целым.

В этой главе описывается реализация, а также программный интерфейс и функции языка запросов для работы с данными больших объектов Postgres Pro. В примерах в этой главе будет использоваться библиотека libpq для языка C, но та же функциональность поддерживается и другими программными интерфейсами Postgres Pro. Другие интерфейсы могут использовать внутри себя интерфейс больших объектов для реализации общего подхода к работе с большими значениями. Здесь это не описывается.

32.1. Введение

Все большие объекты хранятся в одной системной таблице с именем `pg_largeobject`. Для каждого большого объекта также имеется запись в системной таблице `pg_largeobject_metadata`. Большие объекты можно создавать, изменять и удалять, используя API чтения/записи, подобный стандартному API для работы с файлами.

Postgres Pro также поддерживает систему хранения, названную «TOAST», которая автоматически переносит значения, не умещающиеся в одну страницу таблицы, в дополнительную область хранилища. Вследствие этого подсистема больших объектов отчасти оказывается устаревшей. Однако её преимуществом остаётся то, что она позволяет сохранять значения размером до 4 Тбайт, тогда как поля в TOAST ограничиваются 1 Гбайтом. Кроме того, чтение и изменение больших объектов можно выполнять эффективнее по сравнению с полями TOAST, которые при большинстве операций считываются и записываются как единое целое.

32.2. Особенности реализации

Механизм больших объектов разбивает большие объекты на «фрагменты» и сохраняет эти фрагменты в строках таблицы. При произвольном доступе на запись и чтение быстрый поиск нужного фрагмента обеспечивается индексом-B-деревом в этой таблице.

Фрагменты больших объектов не должны быть последовательными. Например, если приложение откроет новый большой объект, переместится к смещению 1000000 байт и запишет несколько байт, это не приведёт к выделению лишнего 1000000 байт в хранилище; записаны будут только фрагменты, покрывающие диапазон собственно записанных байт. Операция чтения, однако, прочитает нули для всех неразмещённых в хранилище байт, предшествующих последнему записанному фрагменту. Это соответствует принятому поведению «разреженных» файлов в файловых системах Unix.

Начиная с PostgreSQL 9.0, для больших объектов назначается владелец и набор прав доступа, которыми можно управлять командами GRANT и REVOKE. Для чтения большого объекта требуются права `SELECT`, а для записи или усечения его — права `UPDATE`. Удалять большой объект, задавать комментарий для него, либо сменять его владельца разрешается только его владельцу (или суперпользователю базы данных). Для совместимости с предыдущими версиями можно скорректировать это поведение, изменив параметр времени выполнения lo_compat_privileges.

32.3. Клиентские интерфейсы

В этом разделе описываются средства, которые предоставляет клиентская библиотека Postgres Pro libpq для обращения к большим объектам. Интерфейс работы с большими объектами Postgres Pro создан по подобию интерфейса файловых систем Unix, так что он включает аналоги функций `open`, `read`, `write`, `lseek` и т. д.

Все операции с большими объектами с применением этих функций *должны* иметь место в блоке транзакции SQL, так как дескрипторы больших объектов актуальны только во время транзакции.

Если при выполнении одной из этих функций происходит ошибка, эта функция возвращает значение, иначе невозможное, обычно 0 или -1. Сообщение, описывающее ошибку, сохраняется в объекте соединения; получить его можно с помощью `PQerrorMessage`.

Клиентские приложения, которые используют эти функции, должны включать заголовочный файл `libpq/libpq-fs.h` и компоноваться с библиотекой libpq.

32.3.1. Создание большого объекта

Функция

```
Oid lo_creat(PGconn *conn, int mode);
```

создаёт новый большой объект. Возвращаемым значением будет OID, назначенный новому объекту, либо `InvalidOid` (ноль) в случае ошибки. Параметр *mode* не используется и игнорируется, начиная с PostgreSQL 8.1; однако для обратной совместимости с более ранними выпусками в нём лучше задать значение `INV_READ`, `INV_WRITE` или `INV_READ | INV_WRITE`. (Эти константы определены в заголовочном файле `libpq/libpq-fs.h`.)

Пример:

```
inv_oid = lo_creat(conn, INV_READ|INV_WRITE);
```

Функция

```
Oid lo_create(PGconn *conn, Oid lobjId);
```

также создаёт новый большой объект. В *lobjId* можно задать назначаемый ему OID; при этом произойдёт ошибка, если этот OID уже присвоен какому-либо большому объекту. Если в *lobjId* передаётся `InvalidOid` (ноль), `lo_create` присваивает большому объекту свободный OID (так же, как и `lo_creat`). Возвращаемым значением будет OID, назначенный новому большому объекту, либо `InvalidOid` (ноль) в случае ошибки.

Функция `lo_create` появилась в PostgreSQL 8.1; если попытаться выполнить её с сервером более старой версии, произойдёт ошибка и будет возвращено `InvalidOid`.

Пример:

```
inv_oid = lo_create(conn, desired_oid);
```

32.3.2. Импорт большого объекта

Чтобы импортировать в качестве большого объекта файл операционной системы, вызовите

```
Oid lo_import(PGconn *conn, const char *filename);
```

В *filename* задаётся имя файла в операционной системе, который будет импортирован как большой объект. Возвращаемым значением будет OID, назначенный новому большому объекту, либо `InvalidOid` (ноль) в случае ошибки. Заметьте, что этот файл читает библиотека клиентского интерфейса, а не сервер; таким образом, он должен существовать в файловой системе на стороне клиента и быть доступным для чтения клиентскому приложению.

Функция

```
Oid lo_import_with_oid(PGconn *conn, const char *filename, Oid lobjId);
```

также импортирует новый большой объект. В *lobjId* можно задать назначаемый ему OID; при этом произойдёт ошибка, если этот OID уже присвоен какому-либо большому объекту. Если в *lobjId* передаётся `InvalidOid` (ноль), `lo_import_with_oid` присваивает большому объекту свободный OID (так же, как и `lo_import`). Возвращаемым значением будет OID, назначенный новому большому объекту, либо `InvalidOid` (ноль) в случае ошибки.

Функция `lo_import_with_oid` появилась в PostgreSQL 8.4 и вызывает внутри `lo_create`, появившуюся в 8.1; если попытаться выполнить её с сервером версии 8.0 или ранней, она завершится ошибкой и возвратит `InvalidOid`.

32.3.3. Экспорт большого объекта

Чтобы экспортировать большой объект в файл операционной системы, вызовите

```
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
```

В аргументе *lobjId* задаётся OID экспортируемого большого объекта, а в аргументе *filename* задаётся имя файла в операционной системе. Заметьте, что файл записывается библиотекой клиентского интерфейса, а не сервером. Возвращает 1 при успешном выполнении, -1 при ошибке.

32.3.4. Открытие существующего большого объекта

Чтобы открыть существующий большой объект для чтения или записи, вызовите

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
```

В аргументе *lobjId* задаётся OID открываемого большого объекта. Биты в аргументе *mode* определяют, открывается ли файл для чтения (INV_READ), для записи (INV_WRITE), либо для чтения/записи. (Эти константы определяются в заголовочном файле libpq/libpq-fs.h.) Функция lo_open возвращает дескриптор большого объекта (неотрицательный) для последующего использования в функциях lo_read, lo_write, lo_lseek, lo_lseek64, lo_tell, lo_tell64, lo_truncate, lo_truncate64 и lo_close. Этот дескриптор актуален только до завершения текущей транзакции. В случае ошибки возвращается -1.

В настоящее время сервер не различает режимы INV_WRITE и INV_READ | INV_WRITE: с таким дескриптором можно читать данные в любом случае. Однако есть значительное отличие этих режимов от одиночного INV_READ: с дескриптором INV_READ записывать данные нельзя, а данные, считываемые через него, будут отражать содержимое большого объекта в снимке транзакции, который был активен при выполнении lo_open, то есть не будут включать изменения, произведённые позже этой или другими транзакциями. При чтении с дескриптором INV_WRITE возвращаются данные, отражающие все изменения, произведённые другими зафиксированными транзакциями, а также текущей транзакцией. Это подобно различиям режимов REPEATABLE READ и READ COMMITTED для обычных команд SQL SELECT.

Пример:

```
inv_fd = lo_open(conn, inv_oid, INV_READ|INV_WRITE);
```

32.3.5. Запись данных в большой объект

Функция

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

записывает *len* байт из буфера *buf* (который должен иметь размер *len*) в дескриптор большого объекта *fd*. В *fd* должно передаваться значение, возвращённое предыдущим вызовом lo_open. Возвращает эта функция число фактически записанных байт (в текущей реализации это всегда *len*, если только не произошла ошибка). В случае ошибки возвращается значение -1.

Хотя параметр *len* объявлен как size_t, эта функция не принимает значение длины, превышающее INT_MAX. На практике всё равно лучше передавать данные фрагментами не больше нескольких мегабайт.

32.3.6. Чтение данных из большого объекта

Функция

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

читает до *len* байт из дескриптора большого объекта *fd* в буфер *buf* (который должен иметь размер *len*). В *fd* должно передаваться значение, возвращённое предыдущим вызовом lo_open. Возвращает эта функция число фактически прочитанных байт; это число должно быть меньше *len*, если при чтении был достигнут конец объекта. В случае ошибки возвращается -1.

Хотя параметр *len* объявлен как size_t, эта функция не принимает значение длины, превышающее INT_MAX. На практике всё равно лучше передавать данные фрагментами не больше нескольких мегабайт.

32.3.7. Перемещение в большом объекте

Чтобы изменить текущее положение чтения или записи, связанное с дескриптором большого объекта, вызовите

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
```

Эта функция перемещает указатель текущего положения для дескриптора большого объекта *fd* в новое положение, заданное аргументом *offset*. Для аргумента *whence* задаются значения SEEK_SET (перемещение от начала объекта), SEEK_CUR (перемещение от текущего положения) и SEEK_END (перемещение от конца объекта). Возвращает эта функция новое положение указателя, либо -1 в случае ошибки.

Оперируя с большими объектами, размер которых превышает 2 ГБ, используйте

```
pg_int64 lo_lseek64(PGconn *conn, int fd, pg_int64 offset, int whence);
```

Эта функция действует так же, как и lo_lseek, но может принять значение *offset*, превышающее 2 ГБ, и/или вернуть результат, превышающий 2 ГБ. Заметьте, что если новое положение указателя оказывается за границей в 2ГБ, функция lo_lseek выдаёт ошибку.

Функция lo_lseek64 появилась в PostgreSQL 9.3. Если попытаться выполнить её с сервером более старой версии, произойдёт ошибка и будет возвращено -1.

32.3.8. Получение текущего положения в большом объекте

Чтобы получить текущее положение чтения или записи для дескриптора большого объекта, вызовите

```
int lo_tell(PGconn *conn, int fd);
```

Если возникает ошибка, возвращается -1.

Оперируя с большими объектами, размер которых может превышать 2 ГБ, используйте

```
pg_int64 lo_tell64(PGconn *conn, int fd);
```

Эта функция действует так же, как lo_tell, но может выдавать результат, превышающий 2 ГБ. Заметьте, что lo_tell выдаёт ошибку, если текущее положение чтения/записи оказывается за границей в 2 ГБ.

Функция lo_tell64 появилась в PostgreSQL 9.3. Если попытаться выполнить её с сервером более старой версии, произойдёт ошибка и будет возвращено -1.

32.3.9. Усечение большого объекта

Чтобы усечь большой объект до требуемой длины, вызовите

```
int lo_truncate(PGconn *conn, int fd, size_t len);
```

Эта функция усекает большой объект с дескриптором *fd* до длины *len*. В *fd* должно передаваться значение, возвращённое предыдущим вызовом lo_open. Если *len* превышает текущую длину большого объекта, большой объект расширяется до заданной длины нулевыми байтами ('\0'). В случае успеха lo_truncate возвращает ноль, а при ошибке возвращается -1.

Положение чтения/записи, связанное с дескриптором *fd*, при этом не меняется.

Хотя параметр *len* объявлен как size_t, lo_truncate не принимает значение длины, превышающее INT_MAX.

Оперируя с большими объектами, размер которых может превышать 2 ГБ, используйте

```
int lo_truncate64(PGconn *conn, int fd, pg_int64 len);
```

Эта функция действует так же, как `lo_truncate`, но может принимать значения *`len`*, превышающие 2 ГБ.

Функция `lo_truncate` появилась в PostgreSQL 8.3; если попытаться выполнить её с сервером более старой версии, произойдёт ошибка и будет возвращено -1.

Функция `lo_truncate64` появилась в PostgreSQL 9.3; если попытаться выполнить её с сервером более старой версии, произойдёт ошибка и будет возвращено -1.

32.3.10. Закрытие дескриптора большого объекта

Дескриптор большого объекта можно закрыть, вызвав

```
int lo_close(PGconn *conn, int fd);
```

Здесь *`fd`* — дескриптор большого объекта, возвращённый функцией `lo_open`. В случае успеха `lo_close` возвращает ноль. При ошибке возвращается -1.

Все дескрипторы больших объектов, остающиеся открытыми в конце транзакции, закрываются автоматически.

32.3.11. Удаление большого объекта

Чтобы удалить большой объект из базы данных, вызовите

```
int lo_unlink(PGconn *conn, Oid lobjId);
```

В аргументе *`lobjId`* задаётся OID большого объекта, который нужно удалить. В случае успеха возвращается 1, а в случае ошибки -1.

32.4. Серверные функции

Функции, предназначенные для работы с большими объектами на стороне сервера из SQL, перечислены в Таблице 32.1.

Таблица 32.1. SQL-ориентированные функции для работы с большими объектами

Функция	Тип результата	Описание	Пример	Результат
`lo_from_bytea( loid oid, string bytea)`	`oid`	Создаёт большой объект, сохраняет в нём данные и возвращает его OID. С `0` система выбирает OID сама.	`lo_from_bytea( 0, '\xffffff00')`	`24528`
`lo_put( loid oid, offset bigint, str bytea)`	`void`	Записывает данные по заданному смещению.	`lo_put(24528, 1, '\xaa')`	
`lo_get( loid oid [, from bigint, for int])`	`bytea`	Извлекает содержимое подстроки.	`lo_get(24528, 0, 3)`	`\xffaaff`

Каждой из клиентских функций, описанных ранее, соответствуют дополнительные функции на стороне сервера; на самом деле, по большей части клиентские функции представляют собой просто интерфейсы к равнозначным серверным функциям. К функциям, которые так же удобно вызывать командами SQL, относятся: `lo_creat`, `lo_create`, `lo_unlink`, `lo_import` и `lo_export`. Ниже приведены примеры их использования:

```
CREATE TABLE image (
    name            text,
    raster          oid
);
```

```
SELECT lo_creat(-1);       -- возвращает OID нового пустого большого объекта

SELECT lo_create(43213);   -- пытается создать большой объект с OID 43213

SELECT lo_unlink(173454);  -- удаляет большой объект с OID 173454

INSERT INTO image (name, raster)
    VALUES ('beautiful image', lo_import('/etc/motd'));

INSERT INTO image (name, raster)  -- то же, что выше, но с предопределённым OID
    VALUES ('beautiful image', lo_import('/etc/motd', 68583));

SELECT lo_export(image.raster, '/tmp/motd') FROM image
    WHERE name = 'beautiful image';
```

Серверные функции `lo_import` и `lo_export` значительно отличаются от их аналогов, выполняемых на стороне клиента. Эти две функции читают и пишут файлы в файловой системе сервера, с правами пользователя, владеющего базами данных. Поэтому использовать их разрешено только суперпользователям. Клиентские функции импорта и экспорта, напротив, читают и пишут файлы в файловой системе клиента, с правами клиентской программы. Для выполнения клиентских функций права суперпользователя не требуются.

Функциональность `lo_read` и `lo_write` также представляется через вызовы на стороне сервера, но имена серверных функций, в отличие от клиентских, не содержат символы подчёркивания. Эти функции нужно вызывать по именам `loread` и `lowrite`.

32.5. Пример программы

В Примере 32.1 представлена пробная программа, демонстрирующая использование интерфейса больших объектов в libpq. Части этой программы закомментированы, но оставлены в тексте для читателя. Эту программу также можно найти в `src/test/examples/testlo.c` в дистрибутиве исходного кода.

Пример 32.1. Пример использования больших объектов с применением libpq

```
/*-------------------------------------------------------------------------
 *
 * testlo.c
 *    test using large objects with libpq
 *
 * Portions Copyright (c) 1996-2017, PostgreSQL Global Development Group
 * Portions Copyright (c) 1994, Regents of the University of California
 *
 *
 * IDENTIFICATION
 *    src/test/examples/testlo.c
 *
 *-------------------------------------------------------------------------
 */
#include <stdio.h>
#include <stdlib.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#include "libpq-fe.h"
#include "libpq/libpq-fs.h"
```

```
#define BUFSIZE         1024

/*
 * importFile -
 *    import file "in_filename" into database as large object "lobjOid"
 *
 */
static Oid
importFile(PGconn *conn, char *filename)
{
    Oid         lobjId;
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * open the file to be read in
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "cannot open unix file\"%s\"\n", filename);
    }

    /*
     * create the large object
     */
    lobjId = lo_creat(conn, INV_READ | INV_WRITE);
    if (lobjId == 0)
        fprintf(stderr, "cannot create large object");

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);

    /*
     * read in from the Unix file and write to the inversion file
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
    {
        tmp = lo_write(conn, lobj_fd, buf, nbytes);
        if (tmp < nbytes)
            fprintf(stderr, "error while reading \"%s\"", filename);
    }

    close(fd);
    lo_close(conn, lobj_fd);

    return lobjId;
}

static void
pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
```

```
    int         nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    nread = 0;
    while (len - nread > 0)
    {
        nbytes = lo_read(conn, lobj_fd, buf, len - nread);
        buf[nbytes] = '\0';
        fprintf(stderr, ">>> %s", buf);
        nread += nbytes;
        if (nbytes <= 0)
            break;              /* no more data? */
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

static void
overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nwritten;
    int         i;

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    for (i = 0; i < len; i++)
        buf[i] = 'X';
    buf[i] = '\0';

    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
        nwritten += nbytes;
        if (nbytes <= 0)
        {
            fprintf(stderr, "\nWRITE FAILED!\n");
            break;
        }
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
```

```
}


/*
 * exportFile -
 *    export large object "lobjOid" to file "out_filename"
 *
 */
static void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * open the large object
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    /*
     * open the file to be written to
     */
    fd = open(filename, O_CREAT | O_WRONLY | O_TRUNC, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "cannot open unix file\"%s\"",
                filename);
    }

    /*
     * read in from the inversion file and write to the Unix file
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
            fprintf(stderr, "error while writing \"%s\"",
                    filename);
        }
    }

    lo_close(conn, lobj_fd);
    close(fd);

    return;
}

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
```

```
}

int
main(int argc, char **argv)
{
    char       *in_filename,
               *out_filename;
    char       *database;
    Oid         lobjOid;
    PGconn     *conn;
    PGresult   *res;

    if (argc != 4)
    {
        fprintf(stderr, "Usage: %s database_name in_filename out_filename\n",
                argv[0]);
        exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * set up the connection
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* check to see that the backend connection was successfully made */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* Set always-secure search path, so malicious users can't take control. */
    res = PQexec(conn,
                 "SELECT pg_catalog.set_config('search_path', '', false)");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "begin");
    PQclear(res);
    printf("importing file \"%s\" ...\n", in_filename);
/*  lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
    if (lobjOid == 0)
        fprintf(stderr, "%s\n", PQerrorMessage(conn));
    else
    {
        printf("\tas large object %u.\n", lobjOid);
```

```
        printf("picking out bytes 1000-2000 of the large object\n");
        pickout(conn, lobjOid, 1000, 1000);

        printf("overwriting bytes 1000-2000 of the large object with X's\n");
        overwrite(conn, lobjOid, 1000, 1000);

        printf("exporting large object to file \"%s\" ...\n", out_filename);
/*      exportFile(conn, lobjOid, out_filename); */
        if (lo_export(conn, lobjOid, out_filename) < 0)
            fprintf(stderr, "%s\n", PQerrorMessage(conn));
    }

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    return 0;
}
```

Глава 33. ECPG — встраиваемый SQL в C

В этой главе описывается встраиваемый SQL для Postgres Pro. Авторы этой разработки: Линус Толке (<linus@epact.se>) и Михаэль Мескес (<meskes@postgresql.org>). Изначально этот проект предназначался для C. Сейчас он также работает с C++, но пока не распознаёт все конструкции C++.

Эта документация не абсолютно полная, но так как этот интерфейс стандартизирован, дополнительные сведения можно почерпнуть во многих ресурсах, посвящённых SQL.

33.1. Концепция

Программа со встраиваемым SQL состоит из кода, написанного на обычном языке программирования, в данном случае C, дополненного командами SQL в специально обозначенных секциях. Чтобы собрать программу, её исходный код (`*.pgc`) сначала нужно пропустить через препроцессор встраиваемого SQL, который превратит её в обычную программу на C (`*.c`), воспринимаемую компилятором C. (Подробнее компиляция и компоновка описывается в Разделе 33.10). Преобразованные приложения ECPG вызывают функции в библиотеке libpq через библиотеку встраиваемого SQL (ecpglib) и взаимодействуют с сервером Postgres Pro по обычному клиент-серверному протоколу.

Встраиваемый SQL имеет ряд преимуществ по сравнению с другими методами вызова команд SQL из кода C. Во-первых, этот механизм берёт на себя заботу о передаче информации через переменные в программе на C. Во-вторых, код SQL в программе проверяется на синтаксическую правильность во время сборки. В-третьих, встраиваемый SQL в C описан стандартом SQL и поддерживается многими другими СУБД SQL. Реализация в Postgres Pro разработана так, чтобы максимально соответствовать этому стандарту, поэтому обычно достаточно легко портировать в Postgres Pro программы с встраиваемым SQL, написанные для других СУБД.

Как уже сказано, программы, написанные для интерфейса встраиваемого SQL, представляют собой обычные программы на C с добавленным специальным кодом, который выполняет действия, связанные с базой данных. Этот специальный код всегда имеет следующую форму:

```
EXEC SQL ...;
```

Такие операторы синтаксически занимают место операторов C. В зависимости от конкретного оператора, они могут размещаться на глобальном уровне или внутри функции. Встраиваемые операторы SQL следуют правилам учёта регистра, принятым в обычном коде SQL, а не в C. Они также допускают вложенные комментарии в стиле C, разрешённые стандартом SQL. Однако остальная часть программы, написанная на C, в соответствии со стандартом C содержать вложенные комментарии не может.

Все встраиваемые SQL-операторы рассматриваются в следующих разделах.

33.2. Управление подключениями к базе данных

В этом разделе описывается, как открывать, закрывать и переключать подключения к базам данных.

33.2.1. Подключение к серверу баз данных

Подключение к базе данных выполняется следующим оператором:

```
EXEC SQL CONNECT TO цель-подключения [AS имя-подключения] [USER имя-пользователя];
```

Цель может задаваться следующими способами:

- *имя_бд*[@*имя_сервера*][:*порт*]
- tcp:postgresql://*имя_сервера*[:*порт*][/*имя_бд*][?*параметры*]
- unix:postgresql://*имя_сервера*[:*порт*][/*имя_бд*][?*параметры*]

- строковая константа SQL, содержащая одну из вышеприведённых записей
- ссылка на символьную переменную, содержащую одну из вышеприведённых записей (см. примеры)
- `DEFAULT`

Если цель подключения задаётся буквально (то есть не через переменную) и значение не заключается в кавычки, регистр в этой строке не учитывается, как в обычном SQL. В этом случае при необходимости также можно заключить в двойные кавычки отдельные параметры. На практике, чтобы не провоцировать ошибки, лучше заключать строку в апострофы, либо передавать её в переменной. С целью подключения `DEFAULT` устанавливается подключение к базе данных по умолчанию с именем пользователя по умолчанию. Другое имя пользователя или имя подключения в этом случае указать нельзя.

Также разными способами можно указать имя пользователя:

- *имя_пользователя*
- *имя_пользователя*/*пароль*
- *имя_пользователя* IDENTIFIED BY *пароль*
- *имя_пользователя* USING *пароль*

В показанных выше строках *имя_пользователя* и *пароль* могут задаваться идентификатором или строковой константой SQL, либо ссылкой на символьную переменную.

Если указание цели подключения включает какие-либо *параметры*, они должны записываться в виде *имя*=*значение* и разделяться амперсандами (`&`). В качестве имён параметров принимаются те же, что поддерживает libpq (см. Подраздел 31.1.2). Перед элементами *имя* или *значение* пробелы игнорируются, но сохраняются внутри или после этих элементов. Заметьте, что записать `&` внутри *значения* нет никакой возможности.

Указание *имя-подключения* применяется, когда в одной программе нужно использовать несколько подключений. Его можно опустить, если программа работает только с одним подключением. Соединение, открытое последним, становится текущим и будет использоваться по умолчанию при выполнении SQL-операторов (это описывается далее в этой главе).

Если к базе данных, которая не приведена в соответствие шаблону безопасного использования схем, имеют доступ недоверенные пользователи, начинайте сеанс с удаления схем, доступных всем для записи, из пути поиска (`search_path`). Например, добавьте `options=-c search_path=` в *options* или выполните `EXEC SQL SELECT pg_catalog.set_config('search_path', '', false);` после подключения. Это касается не только ECPG, но и любых других интерфейсов для выполнения произвольных SQL-команд.

Вот некоторые примеры оператора `CONNECT`:

```
EXEC SQL CONNECT TO mydb@sql.mydomain.com;

EXEC SQL CONNECT TO unix:postgresql://sql.mydomain.com/mydb AS myconnection USER john;

EXEC SQL BEGIN DECLARE SECTION;
const char *target = "mydb@sql.mydomain.com";
const char *user = "john";
const char *passwd = "secret";
EXEC SQL END DECLARE SECTION;
 ...
EXEC SQL CONNECT TO :target USER :user USING :passwd;
/* или EXEC SQL CONNECT TO :target USER :user/:passwd; */
```

В последней форме используется вариант, названный выше ссылкой на символьную переменную. В последующих разделах вы узнаете, как в SQL-операторах можно использовать переменные C, приставляя перед именем двоеточие.

Учтите, что формат цели подключения не описывается в стандарте SQL. Поэтому, если вы хотите разрабатывать переносимые приложения, имеет смысл применить подход, показанный в последнем примере, и сформировать строку подключения отдельно.

33.2.2. Выбор подключения

SQL-операторы в программах со встраиваемым SQL по умолчанию выполняются с текущим подключением, то есть с подключением, которое было открыто последним. Если приложению нужно управлять несколькими подключениями, это можно сделать двумя способами.

Первый вариант — явно выбирать подключение для каждого SQL-оператора, например, так:

```
EXEC SQL AT имя-подключения SELECT ...;
```

Этот вариант хорошо подходит для случаев, когда приложению нужно использовать несколько подключений в смешанном порядке.

Если ваше приложение выполняется в нескольких потоках, они не могут использовать подключение одновременно. Поэтому вы должны либо явно управлять доступом (используя мьютексы), либо использовать отдельные подключения для каждого потока.

Второй вариант — выполнять оператор, переключающий текущее подключение. Этот оператор записывается так:

```
EXEC SQL SET CONNECTION имя-подключения;
```

Этот вариант особенно удобен, когда с одним подключением нужно выполнить несколько операторов.

Следующий пример программы демонстрирует управление несколькими подключениями к базам данных:

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
EXEC SQL END DECLARE SECTION;

int
main()
{
    EXEC SQL CONNECT TO testdb1 AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL CONNECT TO testdb2 AS con2 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL CONNECT TO testdb3 AS con3 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    /* This query would be executed in the last opened database "testdb3". */
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb3)\n", dbname);

    /* Using "AT" to run a query in "testdb2" */
    EXEC SQL AT con2 SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb2)\n", dbname);

    /* Switch the current connection to "testdb1". */
    EXEC SQL SET CONNECTION con1;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb1)\n", dbname);
```

```
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

Этот пример должен вывести следующее:

```
current=testdb3 (should be testdb3)
current=testdb2 (should be testdb2)
current=testdb1 (should be testdb1)
```

33.2.3. Закрытие подключения

Чтобы закрыть подключение, примените следующий оператор:

```
EXEC SQL DISCONNECT [подключение];
```

Подключение можно задать следующими способами:

- *имя-подключения*
- DEFAULT
- CURRENT
- ALL

Если имя подключения не задано, закрывается текущее подключение.

Хорошим стилем считается, когда приложение явно закрывает каждое подключение, которое оно открыло.

33.3. Запуск команд SQL

В приложении со встраиваемым SQL можно запустить любую команду SQL. Ниже приведены несколько примеров, показывающих как это делать.

33.3.1. Выполнение SQL-операторов

Создание таблицы:

```
EXEC SQL CREATE TABLE foo (number integer, ascii char(16));
EXEC SQL CREATE UNIQUE INDEX num1 ON foo(number);
EXEC SQL COMMIT;
```

Добавление строк:

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

Удаление строк:

```
EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;
```

Изменение:

```
EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE number = 9999;
EXEC SQL COMMIT;
```

Операторы SELECT, возвращающие одну строку результата, также могут выполняться непосредственно командой EXEC SQL. Чтобы обработать наборы результатов с несколькими строками, приложение должно использовать курсоры; см. Подраздел 33.3.2 ниже. (В отдельных случаях приложение может выбрать сразу несколько строк в переменную массива; см. Подраздел 33.4.4.3.1.)

Выборка одной строки:

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

Так же можно получить параметр конфигурации командой `SHOW`:

```
EXEC SQL SHOW search_path INTO :var;
```

Идентификаторы вида `:`*`имя`* воспринимаются как *переменные среды*, то есть они ссылаются на переменные программы C. Они рассматриваются в Разделе 33.4.

33.3.2. Использование курсоров

Чтобы получить набор результатов, содержащий несколько строк, приложение должно объявить курсор и выбирать каждую строку через него. Использование курсора подразумевает следующие шаги: объявление курсора, открытие его, выборку строки через курсор, повторение предыдущего шага, и наконец, закрытие курсора.

Выборка с использованием курсоров:

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT number, ascii FROM foo
    ORDER BY ascii;
EXEC SQL OPEN foo_bar;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

Более подробно объявление курсора описывается в DECLARE, а команда `FETCH` описана в FETCH.

> **Примечание**
>
> Команда `DECLARE` в ECPG на самом деле не передаёт этот оператор серверу Postgres Pro. Курсор открывается на сервере (командой сервера `DECLARE`) в момент, когда выполняется команда `OPEN`.

33.3.3. Управление транзакциями

В режиме по умолчанию операторы фиксируются только когда выполняется `EXEC SQL COMMIT`. Интерфейс встраиваемого SQL также поддерживает автофиксацию транзакций (так работает libpq по умолчанию); она включается аргументом командной строки `-t` программы `ecpg` (см. ecpg) либо оператором `EXEC SQL SET AUTOCOMMIT TO ON`. В режиме автофиксации каждая команда фиксируется автоматически, если только она не помещена в явный блок транзакции. Этот режим можно выключить явным образом, выполнив `EXEC SQL SET AUTOCOMMIT TO OFF`.

Поддерживаются следующие команды управления транзакциями:

`EXEC SQL COMMIT`

Зафиксировать текущую транзакцию.

`EXEC SQL ROLLBACK`

Откатить текущую транзакцию.

`EXEC SQL PREPARE TRANSACTION`*`ид_транзакции`*

Подготовить текущую транзакцию для двухфазной фиксации.

`EXEC SQL COMMIT PREPARED`*`ид_транзакции`*

Зафиксировать транзакцию в подготовленном состоянии.

```
EXEC SQL ROLLBACK PREPARED ид_транзакции
```

Откатить транзакцию в подготовленном состоянии.

```
EXEC SQL SET AUTOCOMMIT TO ON
```

Включить режим автофиксации.

```
EXEC SQL SET AUTOCOMMIT TO OFF
```

Отключить режим автофиксации. По умолчанию он отключён.

33.3.4. Подготовленные операторы

Когда значения, передаваемые SQL-оператору, неизвестны во время компиляции, или один и тот же оператор будет использоваться многократно, могут быть полезны подготовленные операторы.

Оператор подготавливается командой `PREPARE`. Вместо значений, которые ещё неизвестны, вставляются местозаполнители «`?`»:

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid, datname FROM pg_database WHERE oid = ?";
```

Если оператор возвращает одну строку, приложение может вызвать `EXECUTE` после `PREPARE` для выполнения этого оператора, указав фактические значения для местозаполнителей в предложении `USING`:

```
EXEC SQL EXECUTE stmt1 INTO :dboid, :dbname USING 1;
```

Если оператор возвращает несколько строк, приложение может использовать курсор, объявленный на базе подготовленного оператора. Чтобы привязать входные параметры, курсор нужно открыть с предложением `USING`:

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid,datname FROM pg_database WHERE oid > ?";
EXEC SQL DECLARE foo_bar CURSOR FOR stmt1;

/* по достижении конца набора результатов прервать цикл while */
EXEC SQL WHENEVER NOT FOUND DO BREAK;

EXEC SQL OPEN foo_bar USING 100;
...
while (1)
{
    EXEC SQL FETCH NEXT FROM foo_bar INTO :dboid, :dbname;
    ...
}
EXEC SQL CLOSE foo_bar;
```

Когда подготовленный оператор больше не нужен, его следует освободить:

```
EXEC SQL DEALLOCATE PREPARE имя;
```

Подробнее оператор `PREPARE` описан в PREPARE. Также обратитесь к Разделу 33.5 за дополнительными сведениями о местозаполнителях и входных параметрах.

33.4. Использование переменных среды

В Разделе 33.3 вы увидели, как можно выполнять SQL-операторы в программе со встраиваемым SQL. Некоторые из этих операторов использовали только фиксированные значения и не давали возможности вставлять в операторы произвольные значения или обрабатывать значения, возвращённые запросом. Операторы такого вида не очень полезны в реальных приложениях. В этом разделе подробно описывается, как можно передавать данные между программой на C и встраиваемыми операторами SQL, используя простой механизм, так называемые *переменные среды*. В программе со встраиваемым SQL мы считаем SQL-операторы *внедрёнными* в код

программы на C, *языке среды*. Таким образом, переменные программы на C называются *переменными среды*.

Ещё один способ передать значения данных между сервером Postgres Pro и приложениями ECPG заключается в использовании дескрипторов SQL, как описано в Разделе 33.7.

33.4.1. Обзор

Передавать данные между программой C и SQL-операторами во встраиваемом SQL очень просто. Вместо того, чтобы вставлять данные в оператор, что влечёт дополнительные усложнения, в частности нужно правильно заключать значения в кавычки, можно просто записать имя переменной C в SQL-операторе, предварив его двоеточием. Например:

```
EXEC SQL INSERT INTO sometable VALUES (:v1, 'foo', :v2);
```

Этот оператор обращается к двум переменным C с именами `v1` и `v2` и также использует обычную строковую константу SQL, показывая тем самым, что можно свободно сочетать разные виды данных.

Этот метод включения переменных C в SQL-операторы работает везде, где SQL-оператор принимает выражение значения.

33.4.2. Секции объявлений

Чтобы передать данные из программы в базу данных, например, в виде параметров запроса, либо получить данные из базы данных в программе, переменные C, которые должны содержать эти данные, нужно объявить в специально помеченных секциях, чтобы препроцессор встраиваемого SQL знал о них.

Секция начинается с:

```
EXEC SQL BEGIN DECLARE SECTION;
```

и заканчивается командой:

```
EXEC SQL END DECLARE SECTION;
```

Между этими строками должны располагаться обычные объявления переменных C, например:

```
int   x = 4;
char  foo[16], bar[16];
```

Как здесь показано, переменной можно присвоить начальное значение. Область видимости переменной определяется расположением секции, в которой она объявляется в программе. Вы также можете объявить переменную следующим образом (при этом неявно создаётся секция объявлений):

```
EXEC SQL int i = 4;
```

Вы можете включать в программу столько секций объявлений, сколько захотите.

Эти объявления выводятся в результирующий файл как объявления обычных переменных C, так что эти переменные не нужно объявлять снова. Переменные, которые не предназначены для использования в командах SQL, можно объявить как обычно вне этих специальных секций.

Определение структуры или объединения тоже должно размещаться в секции `DECLARE`. В противном случае препроцессор не сможет воспринять эти типы, так как не будет знать их определения.

33.4.3. Получение результатов запроса

Теперь вы умеете передавать данные, подготовленные вашей программой, в команду SQL. Но как получить результаты запроса? Для этой цели во встраиваемом SQL есть особые вариации обычных команд `SELECT` и `FETCH`. У этих команд есть специальное предложение `INTO`, определяющее, в какие переменные среды будут помещены получаемые значения. `SELECT` используется для запросов,

возвращающих только одну строку, а FETCH применяется с курсором для запросов, возвращающих несколько строк.

Пример:

```
/*
 * предполагается существование такой таблицы:
 * CREATE TABLE test1 (a int, b varchar(50));
 */

EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL SELECT a, b INTO :v1, :v2 FROM test;
```

Предложение INTO размещается между списком выборки и предложением FROM. Число элементов в списке выборки должно равняться числу элементов в списке после INTO (также называемом целевым списком).

Следующий пример демонстрирует использование команды FETCH:

```
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test;

 ...

do
{
    ...
    EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
    ...
} while (...);
```

Здесь предложение INTO размещается после всех остальных обычных предложений.

33.4.4. Сопоставление типов

Когда приложения ECPG передают данные между сервером Postgres Pro и программой на C, например, получая результаты запроса с сервера или выполняя операторы SQL с входными параметрам, эти данные должны преобразовываться из типов Postgres Pro в типы переменных языка среды (а именно типы языка C) и наоборот. Одно из главных качеств ECPG состоит в том, что в большинстве случаев он делает это автоматически.

В этом отношении можно выделить два вида типов данных. К первому относятся простые типы данных Postgres Pro, такие как integer и text, которые приложение может непосредственно читать и писать. С другими типами данных, такими как timestamp и numeric, можно работать только через специальные функции; см. Подраздел 33.4.4.2.

В Таблице 33.1 показано, как типы данных Postgres Pro соответствуют типам данных C. Когда нужно передать или получить значение определённого типа данных Postgres Pro, вы должны объявить переменную C соответствующего типа C в секции объявлений.

Таблица 33.1. Соответствие между типами данных Postgres Pro и типами переменных C

Тип данных Postgres Pro	Тип переменной среды C
smallint	short
integer	int
bigint	long long int
decimal	decimal[a]
numeric	numeric[a]
real	float
double precision	double
smallserial	short
serial	int
bigserial	long long int
oid	unsigned int
character(*n*), varchar(*n*), text	char[*n*+1], VARCHAR[*n*+1][b]
name	char[NAMEDATALEN]
timestamp	timestamp[a]
interval	interval[a]
date	date[a]
boolean	bool[c]
bytea	char *

[a] С этим типом можно работать только через специальные функции; см. Подраздел 33.4.4.2.
[b] объявляется в ecpglib.h
[c] объявляется в ecpglib.h при отсутствии стандартного объявления

33.4.4.1. Работа с символьными строками

Для обработки типов символьных строк SQL, таких как varchar и text, предлагаются два варианта объявления переменных среды.

Первый способ заключается в использовании char[], массива char, как чаще всего и представляются символьные данные в C.

```
EXEC SQL BEGIN DECLARE SECTION;
    char str[50];
EXEC SQL END DECLARE SECTION;
```

Заметьте, что о длине этого массива вы должны подумать сами. Если вы укажете данную переменную среды C в качестве целевой переменной запроса, возвращающего строку длиннее 49 символов, произойдёт переполнение буфера.

В качестве другого подхода можно использовать специальный тип VARCHAR, представленный в ECPG. Определение массива типа VARCHAR преобразуется в структуру (struct) с собственным именем для каждой переменной. Объявление вида:

```
VARCHAR var[180];
```

преобразуется в:

```
struct varchar_var { int len; char arr[180]; } var;
```

Член структуры arr содержит строку, включающую завершающий нулевой байт. Таким образом, чтобы сохранить строку в переменной типа VARCHAR, эта переменная должна быть объявлена с длиной, учитывающей завершающий нулевой байт. Член структуры len содержит длину строки, сохранённой в arr, без завершающего нулевого байта. Когда на вход запросу подаётся переменная среды C, у которой strlen(arr) отличается от len, применяется наименьшее значение.

VARCHAR можно записать в верхнем или нижнем регистре, но не в смешанном.

Переменные char и VARCHAR также могут содержать значения других типов SQL в их строковом представлении.

33.4.4.2. Обработка специальных типов данных

ECPG представляет некоторые особые типы, которые должны помочь вам легко оперировать некоторыми специальными типами данных Postgres Pro. В частности, в нём реализована поддержка типов numeric, decimal, date, timestamp и interval. Для этих типов нельзя подобрать полезное соответствие с примитивными типами среды (например, int, long long int или char[]), так как они имеют сложную внутреннюю структуру. Приложения, работающие с этими типами, должны объявлять переменные особых типов и работать с ними, применяя функции из библиотеки pgtypes. Эта библиотека, подробно описанная в Разделе 33.6 содержит базовые функции для оперирования этими типами, чтобы вам не требовалось, например, передавать запрос SQL-серверу, когда нужно просто добавить интервал к значению времени.

Эти особые типы данных описаны в следующих подразделах. Чтобы подробнее узнать о функциях в библиотеке pgtypes, обратитесь к Разделу 33.6.

33.4.4.2.1. timestamp, date

Для работы с переменными timestamp в приложении ECPG применяется следующая схема.

Сначала в программу нужно включить заголовочный файл, чтобы получить определение типа timestamp:

```
#include <pgtypes_timestamp.h>
```

Затем объявите в секции объявлений переменную типа timestamp:

```
EXEC SQL BEGIN DECLARE SECTION;
timestamp ts;
EXEC SQL END DECLARE SECTION;
```

Прочитав значение в эту переменную, выполняйте действия с ним, используя функции в библиотеке pgtypes. В следующем примере значение timestamp преобразуется в текстовый вид (ASCII) с помощью функции PGTYPEStimestamp_to_asc():

```
EXEC SQL SELECT now()::timestamp INTO :ts;

printf("ts = %s\n", PGTYPEStimestamp_to_asc(ts));
```

Этот пример выведет такой результат:

```
ts = 2010-06-27 18:03:56.949343
```

Таким же образом можно работать и с типом DATE. В программу нужно включить pgtypes_date.h, объявить переменную типа date, и затем можно будет преобразовать значение DATE в текстовый вид, используя функцию PGTYPESdate_to_asc(). Чтобы подробнее узнать о функциях в библиотеке pgtypes, обратитесь к Разделу 33.6.

33.4.4.2.2. interval

Принцип работы с типом interval тот же, что и с типами timestamp и date, однако для значения типа interval нужно явно выделить память. Другими словами, блок памяти для этой переменной должен размещаться в области кучи, а не в стеке.

Пример программы:

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_interval.h>
```

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    interval *in;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    in = PGTYPESinterval_new();
    EXEC SQL SELECT '1 min'::interval INTO :in;
    printf("interval = %s\n", PGTYPESinterval_to_asc(in));
    PGTYPESinterval_free(in);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

33.4.4.2.3. numeric, decimal

Типы `numeric` и `decimal` обрабатываются так же, как и тип `interval`: вы должны определить указатель, выделить некоторое пространство памяти в куче и обращаться к переменной, используя функции в библиотеке pgtypes. Чтобы подробнее узнать о функциях в библиотеке pgtypes, обратитесь к Разделу 33.6.

Для типа `decimal` никакие специальные функции не реализованы. Для дальнейшей обработки приложение должно преобразовать его в переменную `numeric`, применив функцию из библиотеки pgtypes.

Следующий пример демонстрирует работу с переменными типов `numeric` и `decimal`.

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_numeric.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    numeric *num;
    numeric *num2;
    decimal *dec;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    num = PGTYPESnumeric_new();
    dec = PGTYPESdecimal_new();

    EXEC SQL SELECT 12.345::numeric(4,2), 23.456::decimal(4,2) INTO :num, :dec;

    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 0));
    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 1));
    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 2));
```

```
    /* Преобразовать decimal в numeric, чтобы вывести десятичное значение. */
    num2 = PGTYPESnumeric_new();
    PGTYPESnumeric_from_decimal(dec, num2);

    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 0));
    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 1));
    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 2));

    PGTYPESnumeric_free(num2);
    PGTYPESdecimal_free(dec);
    PGTYPESnumeric_free(num);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

33.4.4.3. Переменные среды для непримитивных типов

В качестве переменных среды также можно использовать массивы, определения типов, структуры и указатели.

33.4.4.3.1. Массивы

Для применения массивов в качестве переменных среды есть два варианта использования. Во-первых, в массиве `char[]` или `VARCHAR[]` можно сохранить текстовую строку, как рассказывалось в Подразделе 33.4.4.1. Во-вторых, в массив можно получить несколько строк из результата запроса, не используя курсор. Чтобы не применяя массивы, обработать результат запроса, состоящий из нескольких строк, нужно использовать курсор и команду `FETCH`. Но с переменными-массивами несколько строк можно получить сразу. Длина определяемого массива должна быть достаточной для размещения всех строк, иначе скорее всего произойдёт переполнение буфера.

Следующий пример сканирует системную таблицу `pg_database` и показывает все OID и имена доступных баз данных:

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int dbid[8];
    char dbname[8][16];
    int i;
EXEC SQL END DECLARE SECTION;

    memset(dbname, 0, sizeof(char)* 16 * 8);
    memset(dbid, 0, sizeof(int) * 8);

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    /* Получить в массивы сразу несколько строк. */
    EXEC SQL SELECT oid,datname INTO :dbid, :dbname FROM pg_database;

    for (i = 0; i < 8; i++)
        printf("oid=%d, dbname=%s\n", dbid[i], dbname[i]);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

Этот пример выводит следующий результат. (Точные значения зависят от локальных обстоятельств.)

```
oid=1, dbname=template1
oid=11510, dbname=template0
oid=11511, dbname=postgres
oid=313780, dbname=testdb
oid=0, dbname=
oid=0, dbname=
oid=0, dbname=
```

33.4.4.3.2. Структуры

Для получения значений сразу нескольких столбцов можно применить структуру, имена членов которой совпадают с именами столбцов результата запроса. Структура позволяет обрабатывать значения нескольких столбцов в одной переменной среды C.

Следующий пример получает значения OID, имена и размеры имеющихся баз данных из системной таблицы `pg_database`, используя при этом функцию `pg_database_size()`. В этом примере переменная типа структуры `dbinfo_t` с членами, имена которых соответствуют именам всех столбцов результата `SELECT`, применяется для получения одной строки результата без вовлечения в оператор `FETCH` нескольких переменных среды.

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef struct
    {
       int oid;
       char datname[65];
       long long int size;
    } dbinfo_t;

    dbinfo_t dbval;
EXEC SQL END DECLARE SECTION;

    memset(&dbval, 0, sizeof(dbinfo_t));

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid) AS size
 FROM pg_database;
    EXEC SQL OPEN cur1;

    /* по достижении конца набора результатов прервать цикл while */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        /* Выбрать несколько столбцов в одну структуру. */
        EXEC SQL FETCH FROM cur1 INTO :dbval;

        /* Напечатать члены структуры. */
        printf("oid=%d, datname=%s, size=%lld\n", dbval.oid, dbval.datname,
 dbval.size);
    }

    EXEC SQL CLOSE cur1;
```

Этот пример показывает следующий результат. (Точные значения зависят от локальных обстоятельств.)

```
oid=1, datname=template1, size=4324580
oid=11510, datname=template0, size=4243460
oid=11511, datname=postgres, size=4324580
```

```
oid=313780, datname=testdb, size=8183012
```

Переменные среды типа структуры «вбирают в себя» столько столбцов, сколько полей содержит структура. Значения дополнительных столбцов можно присвоить другим переменным среды. Например, приведённую выше программу можно видоизменить следующим образом, разместив переменную `size` вне структуры:

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef struct
    {
       int oid;
       char datname[65];
    } dbinfo_t;

    dbinfo_t dbval;
    long long int size;
EXEC SQL END DECLARE SECTION;

    memset(&dbval, 0, sizeof(dbinfo_t));

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid) AS size
 FROM pg_database;
    EXEC SQL OPEN cur1;

    /* по достижении конца набора результатов прервать цикл while */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        /* Выбрать несколько столбцов в одну структуру. */
        EXEC SQL FETCH FROM cur1 INTO :dbval, :size;

        /* Напечатать члены структуры. */
        printf("oid=%d, datname=%s, size=%lld\n", dbval.oid, dbval.datname, size);
    }

    EXEC SQL CLOSE cur1;
```

33.4.4.3.3. Определения типов

Чтобы сопоставить новые типы с уже существующими, используйте ключевое слово `typedef`.

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef char mychartype[40];
    typedef long serial_t;
EXEC SQL END DECLARE SECTION;
```

Заметьте, что вы также можете написать:

```
EXEC SQL TYPE serial_t IS long;
```

Это объявление не обязательно должно находиться в секции объявлений.

33.4.4.3.4. Указатели

Вы можете объявлять указатели на самые распространённые типы. Учтите, однако, что указатели нельзя использовать в качестве целевых переменных запросов без автовыделения. За дополнительными сведениями об автовыделении обратитесь к Разделу 33.7.

```
EXEC SQL BEGIN DECLARE SECTION;
    int   *intp;
    char **charp;
```

```
EXEC SQL END DECLARE SECTION;
```

33.4.5. Обработка непримитивных типов данных SQL

В этом разделе описывается как работать с нескалярными и пользовательскими типами уровня SQL в приложениях ECPG. Заметьте, что этот подход отличается от использования переменных непримитивных типов, описанного в предыдущем разделе.

33.4.5.1. Массивы

Многомерные массивы уровня SQL в ECPG напрямую не поддерживаются, но одномерные массивы уровня SQL могут быть сопоставлены с переменными-массивами среды C и наоборот. Однако учтите, что когда создаётся оператор, ecpg не знает типов столбцов, поэтому не может проверить, вводится ли массив C в соответствующий массив уровня SQL. Обрабатывая результат SQL-оператора, ecpg имеет необходимую информацию и таким образом может убедиться, что с обеих сторон массивы.

Если запрос обращается к отдельным *элементам* массива, это избавляет от необходимости применять массивы в ECPG. В этом случае следует использовать переменную среды C, имеющую тип, который можно сопоставить типу элемента. Например, если типом столбца является массив `integer`, можно использовать переменную среды типа `int`. Аналогично, если тип элемента — `varchar` или `text`, можно использовать переменную типа `char[]` или `VARCHAR[]`.

Предположим, что у нас есть таблица:

```
CREATE TABLE t3 (
    ii integer[]
);

testdb=> SELECT * FROM t3;
     ii
-------------
 {1,2,3,4,5}
(1 row)
```

Следующая программа получает 4-ый элемент массива и сохраняет его в переменной среды C, имеющей тип `int`:

```
EXEC SQL BEGIN DECLARE SECTION;
int ii;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[4] FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :ii ;
    printf("ii=%d\n", ii);
}

EXEC SQL CLOSE cur1;
```

Этот пример выводит следующий результат:

```
ii=4
```

Чтобы сопоставить несколько элементов массива с несколькими элементами переменной-массивом среды, каждый элемент массива SQL нужно по отдельности связать с каждым элементом массива среды, например:

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[1], ii[2], ii[3], ii[4] FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :ii_a[0], :ii_a[1], :ii_a[2], :ii_a[3];
    ...
}
```

Ещё раз обратите внимание, что в этом случае вариант

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* НЕПРАВИЛЬНО */
    EXEC SQL FETCH FROM cur1 INTO :ii_a;
    ...
}
```

не будет работать корректно, так как столбец с типом массива нельзя напрямую сопоставить с переменной-массивом среды.

Можно также применить обходное решение — хранить массивы в их внешнем строковом представлении в переменных среды типа `char[]` или `VARCHAR[]`. Более подробно это представление описывается в Подразделе 8.15.2. Заметьте, это означает, что с таким массивом в программе нельзя будет работать естественным образом (без дополнительного разбора текстового представления).

33.4.5.2. Составные типы

Составные типы в ECPG напрямую не поддерживаются, но есть простое обходное решение. Для решения этой проблемы можно применить те же подходы, что были описаны выше для массивов: обращаться к каждому атрибуту по отдельности или использовать внешнее строковое представление.

Для следующих примеров предполагается, что существует такой тип и таблица:

```
CREATE TYPE comp_t AS (intval integer, textval varchar(32));
CREATE TABLE t4 (compval comp_t);
INSERT INTO t4 VALUES ( (256, 'Postgres Pro') );
```

Самое очевидное решение заключается в обращении к каждому атрибуту по отдельности. Следующая программа получает данные из тестовой таблицы, выбирая атрибуты типа `comp_t` по одному:

```
EXEC SQL BEGIN DECLARE SECTION;
int intval;
varchar textval[33];
```

```
EXEC SQL END DECLARE SECTION;

/* Указать каждый элемент столбца составного типа в списке SELECT. */
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* Выбрать каждый элемент столбца составного типа в переменную среды C. */
    EXEC SQL FETCH FROM cur1 INTO :intval, :textval;

    printf("intval=%d, textval=%s\n", intval, textval.arr);
}

EXEC SQL CLOSE cur1;
```

В развитие этого примера, переменные среды, в которые помещаются результаты команды `FETCH`, можно собрать в одну структуру. Подробнее переменные среды в форме структуры описываются в Подразделе 33.4.4.3.2. Чтобы перейти к структуре, пример можно изменить как показано ниже. Переменные среды, `intval` и `textval`, становятся членами структуры `comp_t`, и эта структура указывается в команде `FETCH`.

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct
{
    int intval;
    varchar textval[33];
} comp_t;

comp_t compval;
EXEC SQL END DECLARE SECTION;

/* Поместить каждый элемент составного типа в список SELECT. */
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* Поместить все значения списка SELECT в одну структуру. */
    EXEC SQL FETCH FROM cur1 INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);
}

EXEC SQL CLOSE cur1;
```

Хотя в команде `FETCH` используется структура, имена атрибутов в предложении `SELECT` задаются по одному. Это можно дополнительно улучшить, написав `*`, что будет обозначать все атрибуты значения составного типа.

```
...
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).* FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;
```

```
while (1)
{
    /* Выбрать все значения в списке SELECT в одну структуру. */
    EXEC SQL FETCH FROM cur1 INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);
}
...
```

Таким образом составные типы можно сопоставить со структурами практически прозрачно, хотя ECPG и не понимает составные типы.

Наконец, также можно сохранить значения составного типа в их внешнем строковом представлении в переменных среды типа `char[]` или `VARCHAR[]`. Однако при таком подходе нет простой возможности обращаться из программы к полям значения.

33.4.5.3. Пользовательские базовые типы

Определяемые пользователем базовые типы не поддерживаются ECPG напрямую. Для них можно использовать внешнее строковое представление и переменные среды типа `char[]` или `VARCHAR[]`, и это решение действительно будет подходящим и достаточным для большинства типов.

Следующий фрагмент кода демонстрирует использование типа данных `complex` из примера в Разделе 35.11. Внешнее строковое представление этого типа имеет форму `(%f,%f)` и определено в функциях `complex_in()` и `complex_out()` в Разделе 35.11. Следующий пример вставляет значения комплексного типа `(1,1)` и `(3,3)` в столбцы `a` и `b`, а затем выбирает их из таблицы.

```
EXEC SQL BEGIN DECLARE SECTION;
    varchar a[64];
    varchar b[64];
EXEC SQL END DECLARE SECTION;

    EXEC SQL INSERT INTO test_complex VALUES ('(1,1)', '(3,3)');

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT a, b FROM test_complex;
    EXEC SQL OPEN cur1;

    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        EXEC SQL FETCH FROM cur1 INTO :a, :b;
        printf("a=%s, b=%s\n", a.arr, b.arr);
    }

    EXEC SQL CLOSE cur1;
```

Этот пример выводит следующий результат:

```
a=(1,1), b=(3,3)
```

Другое обходное решение состоит в том, чтобы избегать прямого использования пользовательских типов в ECPG, а вместо этого создать функцию или приведение, выполняющее преобразование между пользовательским типом и примитивным типом, который может обработать ECPG. Заметьте, однако, что приведения типов, особенно неявные, нужно добавлять в систему типов очень осторожно.

Например:

```
CREATE FUNCTION create_complex(r double, i double) RETURNS complex
LANGUAGE SQL
IMMUTABLE
```

```
AS $$ SELECT $1 * complex '(1,0')' + $2 * complex '(0,1)' $$;
```

После такого определения следующий код

```
EXEC SQL BEGIN DECLARE SECTION;
double a, b, c, d;
EXEC SQL END DECLARE SECTION;

a = 1;
b = 2;
c = 3;
d = 4;

EXEC SQL INSERT INTO test_complex VALUES (create_complex(:a, :b),
 create_complex(:c, :d));
```

будет работать так же, как

```
EXEC SQL INSERT INTO test_complex VALUES ('(1,2)', '(3,4)');
```

33.4.6. Индикаторы

Приведённые выше примеры никак не обрабатывали значения NULL. На самом деле, в примерах с извлечением данных возникнет ошибка, если они выберут из базы данных значение NULL. Чтобы можно было передавать значения NULL в базу данных или получать их из базы данных, вы должны добавить объявление второй переменной среды C для каждой переменной C, содержащей данные. Эта вторая переменная среды C называется *индикатором* и содержит флаг, показывающий, что в данных передаётся NULL, и при этом значение основной переменной среды C игнорируется. Следующий пример демонстрирует правильную обработку значений NULL:

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR val;
int val_ind;
EXEC SQL END DECLARE SECTION:

 ...

EXEC SQL SELECT b INTO :val :val_ind FROM test1;
```

Переменная индикатора `val_ind` будет равна нулю, если значение не NULL, или отрицательному числу, если NULL.

Индикатор выполняет ещё одну функцию: если он содержит положительное число, это означает, что значение не NULL, но оно было обрезано, когда сохранялось в переменной среды C.

Если препроцессору `ecpg` передаётся аргумент `-r no_indicator`, он работает в режиме «без индикатора». В этом режиме, если переменная индикатора не определена, значения NULL обозначаются (при вводе и выводе) для символьных строк пустой строкой, а для целочисленных типов наименьшим возможным значением этого типа (например, `INT_MIN` для `int`).

33.5. Динамический SQL

Во многих случаях конкретные SQL-операторы, которые должно выполнять приложение, известны в момент написания приложения. В некоторых случаях, однако, SQL-операторы формируются во время выполнения или поступают из внешнего источника. В этих случаях SQL-операторы нельзя внедрить непосредственно в исходный код C, но есть средство, позволяющее вызывать произвольные SQL-операторы, передаваемые в строковой переменной.

33.5.1. Выполнение операторов без набора результатов

Самый простой способ выполнить произвольный SQL-оператор — применить команду `EXECUTE IMMEDIATE`. Например:

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "CREATE TABLE test1 (...);";
EXEC SQL END DECLARE SECTION;

EXEC SQL EXECUTE IMMEDIATE :stmt;
```

`EXECUTE IMMEDIATE` можно применять для SQL-операторов, которые не возвращают набор результатов (например, DDL, `INSERT`, `UPDATE`, `DELETE`). Выполнять операторы, которые получают данные, (например, `SELECT`) таким образом нельзя. Как выполнять такие операторы, рассказывается в следующем разделе.

33.5.2. Выполнение оператора с входными параметрами

Более эффективно выполнять произвольный SQL-оператор можно, подготовив его один раз, а затем запуская подготовленный оператор столько, сколько нужно. Также можно подготовить обобщённую версию оператора, а затем выполнять специализированные его версии, подставляя в него параметры. Подготавливая оператор, поставьте знаки вопроса там, где позже хотите подставить параметры. Например:

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "INSERT INTO test1 VALUES(?, ?);";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt USING 42, 'foobar';
```

Когда подготовленный оператор больше не нужен, его следует освободить:

```
EXEC SQL DEALLOCATE PREPARE имя;
```

33.5.3. Выполнение оператора с набором результатов

Для выполнения SQL-оператора с одной строкой результата можно применить команду `EXECUTE`. Чтобы сохранить результат, добавьте предложение `INTO`.

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "SELECT a, b, c FROM test1 WHERE a > ?";
int v1, v2;
VARCHAR v3[50];
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt INTO :v1, :v2, :v3 USING 37;
```

Команда `EXECUTE` может содержать предложение `INTO` и/или предложение `USING`, либо не содержать ни того, ни другого.

Если ожидается, что запрос вернёт более одной строки результата, следует применять курсор, как показано в следующем примере. (Подробно курсоры описываются в Подразделе 33.3.2.)

```
EXEC SQL BEGIN DECLARE SECTION;
char dbaname[128];
char datname[128];
char *stmt = "SELECT u.usename as dbaname, d.datname "
             "  FROM pg_database d, pg_user u "
             "  WHERE d.datdba = u.usesysid";
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO testdb AS con1 USER testuser;
```

```
EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

EXEC SQL PREPARE stmt1 FROM :stmt;

EXEC SQL DECLARE cursor1 CURSOR FOR stmt1;
EXEC SQL OPEN cursor1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH cursor1 INTO :dbaname,:datname;
    printf("dbaname=%s, datname=%s\n", dbaname, datname);
}

EXEC SQL CLOSE cursor1;

EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
```

33.6. Библиотека pgtypes

Библиотека pgtypes сопоставляет типы базы данных Postgres Pro с их эквивалентами в C, которые можно использовать в программах на C. Она также предлагает функции для выполнения простых вычислений с этими типами в C, то есть без помощи сервера Postgres Pro. Рассмотрите следующий пример:

```
EXEC SQL BEGIN DECLARE SECTION;
   date date1;
   timestamp ts1, tsout;
   interval iv1;
   char *out;
EXEC SQL END DECLARE SECTION;

PGTYPESdate_today(&date1);
EXEC SQL SELECT started, duration INTO :ts1, :iv1 FROM datetbl WHERE d=:date1;
PGTYPEStimestamp_add_interval(&ts1, &iv1, &tsout);
out = PGTYPEStimestamp_to_asc(&tsout);
printf("Started + duration: %s\n", out);
PGTYPESchar_free(out);
```

33.6.1. Символьные строки

Некоторые функции, в частности `PGTYPESnumeric_to_asc`, возвращают указатель на строку в выделенной для неё памяти. Их результаты должны освобождаться функцией `PGTYPESchar_free`, а не `free`. (Это важно только в Windows, где выделение и освобождение памяти в определённых случаях должно производиться одной библиотекой.)

33.6.2. Тип numeric

Тип numeric позволяет производить вычисления с произвольной точностью. Эквивалентный ему тип на сервере Postgres Pro описан в Разделе 8.1. Ввиду того, что переменная имеет произвольную точность, она должна расширяться и сжиматься динамически. Поэтому такие переменные можно создавать только в области кучи, используя функции `PGTYPESnumeric_new` и `PGTYPESnumeric_free`. Тип decimal подобен numeric, но имеет ограниченную точность, и поэтому может размещаться и в области кучи, и в стеке.

Для работы с типом numeric можно использовать следующие функции:

PGTYPESnumeric_new

Запрашивает указатель на новую переменную, размещённую в памяти.

```
numeric *PGTYPESnumeric_new(void);
```

PGTYPESnumeric_free

Освобождает переменную типа numeric, высвобождая всю её память.

```
void PGTYPESnumeric_free(numeric *var);
```

PGTYPESnumeric_from_asc

Разбирает числовой тип из строковой записи.

```
numeric *PGTYPESnumeric_from_asc(char *str, char **endptr);
```

Допускаются в частности следующие форматы: `-2`, `.794`, `+3.44`, `592.49E07` и `-32.84e-4`. Если значение удаётся разобрать успешно, возвращается действительный указатель, в противном случае указатель NULL. На данный момент ECPG всегда разбирает строку до конца, так что эта функция не может вернуть адрес первого недопустимого символа в `*endptr`. Поэтому в `endptr` свободно можно передать NULL.

PGTYPESnumeric_to_asc

Возвращает указатель на строку, выделенную функцией `malloc` и содержащую строковое представление значения `num` числового типа.

```
char *PGTYPESnumeric_to_asc(numeric *num, int dscale);
```

Числовое значение будет выводиться с заданным в `dscale` количеством цифр после запятой, округлённое при необходимости. Результат нужно освободить функцией `PGTYPESchar_free()`.

PGTYPESnumeric_add

Суммирует две числовые переменные и возвращает результат в третьей.

```
int PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result);
```

Эта функция суммирует переменные `var1` и `var2` в результирующую переменную `result`. Функция возвращает 0 в случае успеха и -1 при ошибке.

PGTYPESnumeric_sub

Вычисляет разность двух числовых переменных и возвращает результат в третьей.

```
int PGTYPESnumeric_sub(numeric *var1, numeric *var2, numeric *result);
```

Эта функция вычитает переменную `var2` из `var1`. Результат операции помещается в переменную `result`. Функция возвращает 0 в случае успеха и -1 при ошибке.

PGTYPESnumeric_mul

Перемножает две числовые переменные и возвращает результат в третьей.

```
int PGTYPESnumeric_mul(numeric *var1, numeric *var2, numeric *result);
```

Эта функция перемножает переменные `var1` и `var2`. Результат операции сохраняется в переменной `result`. Функция возвращает 0 в случае успеха и -1 при ошибке.

PGTYPESnumeric_div

Вычисляет частное двух числовых переменных и возвращает результат в третьей.

```
int PGTYPESnumeric_div(numeric *var1, numeric *var2, numeric *result);
```

Эта функция делит переменную `var1` на `var2`. Результат операции сохраняется в переменной `result`. Функция возвращает 0 в случае успеха и -1 при ошибке.

PGTYPESnumeric_cmp

Сравнивает две числовые переменные.

```
int PGTYPESnumeric_cmp(numeric *var1, numeric *var2)
```

Эта функция производит сравнение двух числовых переменных. При ошибке возвращается INT_MAX. В случае успеха функция возвращает одно из трёх возможных значений:

- 1, если var1 больше var2
- -1, если var1 меньше var2
- 0, если var1 и var2 равны

PGTYPESnumeric_from_int

Преобразует переменную int в переменную numeric.

```
int PGTYPESnumeric_from_int(signed int int_val, numeric *var);
```

Эта функция принимает целочисленную переменную со знаком типа signed int и сохраняет её значение в переменной var типа numeric. Функция возвращает 0 в случае успеха и -1 при ошибке.

PGTYPESnumeric_from_long

Преобразует переменную long int в переменную numeric.

```
int PGTYPESnumeric_from_long(signed long int long_val, numeric *var);
```

Эта функция принимает целочисленную переменную со знаком типа signed long int и сохраняет её значение в переменной var типа numeric. Функция возвращает 0 в случае успеха и -1 при ошибке.

PGTYPESnumeric_copy

Копирует одну числовую переменную в другую.

```
int PGTYPESnumeric_copy(numeric *src, numeric *dst);
```

Эта функция копирует значение переменной, на которую указывает src, в переменную, на которую указывает dst. Она возвращает 0 в случае успеха и -1 при ошибке.

PGTYPESnumeric_from_double

Преобразует переменную типа double в переменную numeric.

```
int  PGTYPESnumeric_from_double(double d, numeric *dst);
```

Эта функция принимает переменную типа double и сохраняет преобразованное значение в переменной, на которую указывает dst. Она возвращает 0 в случае успеха и -1 при ошибке.

PGTYPESnumeric_to_double

Преобразует переменную типа numeric в переменную double.

```
int PGTYPESnumeric_to_double(numeric *nv, double *dp)
```

Эта функция преобразует значение типа numeric переменной, на которую указывает nv, в переменную типа double, на которую указывает dp. Она возвращает 0 в случае успеха и -1 при ошибке, в том числе при переполнении. Если происходит переполнение, в глобальной переменной errno дополнительно устанавливается значение PGTYPES_NUM_OVERFLOW.

PGTYPESnumeric_to_int

Преобразует переменную типа numeric в переменную int.

```
int PGTYPESnumeric_to_int(numeric *nv, int *ip);
```

Эта функция преобразует значение типа numeric переменной, на которую указывает `nv`, в целочисленную переменную, на которую указывает `ip`. Она возвращает 0 в случае успеха и -1 при ошибке, в том числе при переполнении. Если происходит переполнение, в глобальной переменной `errno` дополнительно устанавливается значение `PGTYPES_NUM_OVERFLOW`.

`PGTYPESnumeric_to_long`

Преобразует переменную типа numeric в переменную long.

```
int PGTYPESnumeric_to_long(numeric *nv, long *lp);
```

Эта функция преобразует значение типа numeric переменной, на которую указывает `nv`, в целочисленную переменную типа long, на которую указывает `lp`. Она возвращает 0 в случае успеха и -1 при ошибке, в том числе при переполнении. Если происходит переполнение, в глобальной переменной `errno` дополнительно устанавливается значение `PGTYPES_NUM_OVERFLOW`.

`PGTYPESnumeric_to_decimal`

Преобразует переменную типа numeric в переменную decimal.

```
int PGTYPESnumeric_to_decimal(numeric *src, decimal *dst);
```

Эта функция преобразует значение типа numeric переменной, на которую указывает `src`, в переменную типа decimal, на которую указывает `dst`. Она возвращает 0 в случае успеха и -1 при ошибке, в том числе при переполнении. Если происходит переполнение, в глобальной переменной `errno` дополнительно устанавливается значение `PGTYPES_NUM_OVERFLOW`.

`PGTYPESnumeric_from_decimal`

Преобразует переменную типа decimal в переменную numeric.

```
int PGTYPESnumeric_from_decimal(decimal *src, numeric *dst);
```

Эта функция преобразует значение типа decimal переменной, на которую указывает `src`, в переменную типа numeric, на которую указывает `dst`. Она возвращает 0 в случае успеха и -1 при ошибке. Так как тип decimal реализован как ограниченная версия типа numeric, при таком преобразовании переполнение невозможно.

33.6.3. Тип date

Тип date, реализованный в C, позволяет программам работать с данными типа date в SQL. Соответствующий тип сервера Postgres Pro описан в Разделе 8.5.

Для работы с типом date можно использовать следующие функции:

`PGTYPESdate_from_timestamp`

Извлекает часть даты из значения типа timestamp.

```
date PGTYPESdate_from_timestamp(timestamp dt);
```

Эта функция получает в единственном аргументе значение времени типа timestamp и возвращает извлечённую из него дату.

`PGTYPESdate_from_asc`

Разбирает дату из её текстового представления.

```
date PGTYPESdate_from_asc(char *str, char **endptr);
```

Эта функция получает строку C char* `str` и указатель на строку C char* `endptr`. На данный момент ECPG всегда разбирает строку до конца, так что эта функция не может вернуть адрес первого недопустимого символа в `*endptr`. Поэтому в `endptr` свободно можно передать NULL.

Заметьте, что эта функция всегда подразумевает формат дат MDY (месяц-день-год) и никакой переменной для изменения этого формата в ECPG нет.

Все допустимые форматы ввода перечислены в Таблице 33.2.

Таблица 33.2. Допустимые форматы ввода для `PGTYPESdate_from_asc`

Ввод	Результат
January 8, 1999	January 8, 1999
1999-01-08	January 8, 1999
1/8/1999	January 8, 1999
1/18/1999	January 18, 1999
01/02/03	February 1, 2003
1999-Jan-08	January 8, 1999
Jan-08-1999	January 8, 1999
08-Jan-1999	January 8, 1999
99-Jan-08	January 8, 1999
08-Jan-99	January 8, 1999
08-Jan-06	January 8, 2006
Jan-08-99	January 8, 1999
19990108	ISO 8601; January 8, 1999
990108	ISO 8601; January 8, 1999
1999.008	год и день года
J2451187	Юлианский день
January 8, 99 BC	99 год до нашей эры

`PGTYPESdate_to_asc`

Возвращает текстовое представление переменной типа date.

```
char *PGTYPESdate_to_asc(date dDate);
```

Эта функция получает в качестве единственного параметра дату `dDate` и выводит её в виде `1999-01-18`, то есть в формате `YYYY-MM-DD`. Результат необходимо освободить функцией `PGTYPESchar_free()`.

`PGTYPESdate_julmdy`

Извлекает значения дня, месяца и года из переменной типа date.

```
void PGTYPESdate_julmdy(date d, int *mdy);
```

Эта функция получает дату `d` и указатель на 3 целочисленных значения `mdy`. Имя переменной указывает на порядок значений: в `mdy[0]` записывается номер месяца, в `mdy[1]` — номер дня, а в `mdy[2]` — год.

`PGTYPESdate_mdyjul`

Образует значение даты из массива 3 целых чисел, задающих день, месяц и год даты.

```
void PGTYPESdate_mdyjul(int *mdy, date *jdate);
```

Эта функция получает в первом аргументе массив из 3 целых чисел (`mdy`), а во втором указатель на переменную типа date, в которую будет помещён результат операции.

`PGTYPESdate_dayofweek`

Возвращает число, представляющее день недели для заданной даты.

```
int PGTYPESdate_dayofweek(date d);
```

Эта функция принимает в единственном аргументе переменную `d` типа date и возвращает целое число, выражающее день недели для этой даты.

- 0 — Воскресенье
- 1 — Понедельник
- 2 — Вторник
- 3 — Среда
- 4 — Четверг
- 5 — Пятница
- 6 — Суббота

`PGTYPESdate_today`

Выдаёт текущую дату.

```
void PGTYPESdate_today(date *d);
```

Эта функция получает указатель на переменную (`d`) типа date, в которую будет записана текущая дата.

`PGTYPESdate_fmt_asc`

Преобразует переменную типа date в текстовое представление по маске формата.

```
int PGTYPESdate_fmt_asc(date dDate, char *fmtstring, char *outbuf);
```

Эта функция принимает дату для преобразования (`dDate`), маску формата (`fmtstring`) и строку, в которую будет помещено текстовое представление даты (`outbuf`).

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

В строке формата можно использовать следующие коды полей:

- `dd` — Номер дня в месяце.
- `mm` — Номер месяца в году.
- `yy` — Номер года в виде двух цифр.
- `yyyy` — Номер года в виде четырёх цифр.
- `ddd` — Название дня недели (сокращённое).
- `mmm` — Название месяца (сокращённое).

Все другие символы копируются в выводимую строку 1:1.

В Таблице 33.3 перечислены несколько возможных форматов. Это даёт представление, как можно использовать эту функцию. Все строки вывода даны для одной даты: 23 ноября 1959 г.

Таблица 33.3. Допустимые форматы ввода для `PGTYPESdate_fmt_asc`

Формат	Результат
mmddyy	112359
ddmmyy	231159
yymmdd	591123
yy/mm/dd	59/11/23
yy mm dd	59 11 23
yy.mm.dd	59.11.23
.mm.yyyy.dd.	.11.1959.23.
mmm. dd, yyyy	Nov. 23, 1959

Формат	Результат
mmm dd yyyy	Nov 23 1959
yyyy dd mm	1959 23 11
ddd, mmm. dd, yyyy	Mon, Nov. 23, 1959
(ddd) mmm. dd, yyyy	(Mon) Nov. 23, 1959

PGTYPESdate_defmt_asc

Преобразует строку C char* в значение типа date по маске формата.

```
int PGTYPESdate_defmt_asc(date *d, char *fmt, char *str);
```

Эта функция принимает указатель на переменную типа date (d), в которую будет помещён результат операции, маску формата для разбора даты (fmt) и строку C char*, содержащую текстовое представление даты (str). Ожидается, что текстовое представление будет соответствовать маске формата. Однако это соответствие не обязательно должно быть точным. Данная функция анализирует только порядок элементов и ищет в нём подстроки yy или yyyy, обозначающие позицию года, подстроку mm, обозначающую позицию месяца, и dd, обозначающую позицию дня.

В Таблица 33.4 перечислены несколько возможных форматов. Это даёт представление, как можно использовать эту функцию.

Таблица 33.4. Допустимые форматы ввода для rdefmtdate

Формат	Строка	Результат
ddmmyy	21-2-54	1954-02-21
ddmmyy	2-12-54	1954-12-02
ddmmyy	20111954	1954-11-20
ddmmyy	130464	1964-04-13
mmm.dd.yyyy	MAR-12-1967	1967-03-12
yy/mm/dd	1954, February 3rd	1954-02-03
mmm.dd.yyyy	041269	1969-04-12
yy/mm/dd	In the year 2525, in the month of July, mankind will be alive on the 28th day	2525-07-28
dd-mm-yy	I said on the 28th of July in the year 2525	2525-07-28
mmm.dd.yyyy	9/14/58	1958-09-14
yy/mm/dd	47/03/29	1947-03-29
mmm.dd.yyyy	oct 28 1975	1975-10-28
mmddyy	Nov 14th, 1985	1985-11-14

33.6.4. Тип timestamp

Тип timestamp, реализованный в C, позволяет программам работать с данными типа timestamp в SQL. Соответствующий тип сервера Postgres Pro описан в Разделе 8.5.

Для работы с типом timestamp можно использовать следующие функции:

PGTYPEStimestamp_from_asc

Разбирает значение даты/времени из текстового представления в переменную типа timestamp.

```
timestamp PGTYPEStimestamp_from_asc(char *str, char **endptr);
```

Эта функция получает строку (`str`), которую нужно разобрать, и указатель на строку C char* (`endptr`). На данный момент ECPG всегда разбирает строку до конца, так что эта функция не может вернуть адрес первого недопустимого символа в `*endptr`. Поэтому в `endptr` свободно можно передать NULL.

В случае успеха эта функция возвращает разобранное время, а в случае ошибки возвращается `PGTYPESInvalidTimestamp` и в `errno` устанавливается значение `PGTYPES_TS_BAD_TIMESTAMP`. См. замечание относительно `PGTYPESInvalidTimestamp`.

Вообще вводимая строка может содержать допустимое указание даты, пробельные символы и допустимое указание времени в любом сочетании. Заметьте, что часовые пояса ECPG не поддерживает. Эта функция может разобрать их, но не задействует их в вычислениях как это делает, например, сервер Postgres Pro. Указания часового пояса во вводимой строке просто игнорируются.

В Таблица 33.5 приведены несколько примеров вводимых строк.

Таблица 33.5. Допустимые форматы ввода для `PGTYPEStimestamp_from_asc`

Ввод	Результат
`1999-01-08 04:05:06`	`1999-01-08 04:05:06`
`January 8 04:05:06 1999 PST`	`1999-01-08 04:05:06`
`1999-Jan-08 04:05:06.789-8`	`1999-01-08 04:05:06.789 (указание часового пояса игнорируется)`
`J2451187 04:05-08:00`	`1999-01-08 04:05:00 (указание часового пояса игнорируется)`

`PGTYPEStimestamp_to_asc`

Преобразует значение даты в строку C char*.

```
char *PGTYPEStimestamp_to_asc(timestamp tstamp);
```

Эта функция принимает в качестве единственного аргумента `tstamp` значение типа timestamp и возвращает размещённую в памяти строку, содержащую текстовое представление даты/времени. Результат необходимо освободить функцией `PGTYPESchar_free()`.

`PGTYPEStimestamp_current`

Получает текущее время.

```
void PGTYPEStimestamp_current(timestamp *ts);
```

Эта функция получает текущее время и сохраняет его в переменной типа timestamp, на которую указывает `ts`.

`PGTYPEStimestamp_fmt_asc`

Преобразует переменную типа timestamp в строку C char* по маске формата.

```
int PGTYPEStimestamp_fmt_asc(timestamp *ts, char *output, int str_len, char
 *fmtstr);
```

Эта функция получает в первом аргументе (`ts`) указатель на переменную типа timestamp, а в последующих указатель на буфер вывода (`output`), максимальную длину строки, которую может принять буфер (`str_len`), и маску формата, с которой будет выполняться преобразование (`fmtstr`).

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

В маске формата можно использовать коды формата, перечисленные ниже. Эти же коды принимает функция `strftime` из библиотеки libc. Любые символы, не относящиеся к кодам формата, будут просто скопированы в буфер вывода.

- `%A` — заменяется локализованным представлением полного названия дня недели.
- `%a` — заменяется локализованным представлением сокращённого названия дня недели.
- `%B` — заменяется локализованным представлением полного названия месяца.
- `%b` — заменяется локализованным представлением сокращённого названия месяца.
- `%C` — заменяется столетием (год / 100) в виде десятичного числа; одиночная цифра предваряется нулём.
- `%c` — заменяется локализованным представлением даты и времени.
- `%D` — равнозначно `%m/%d/%y`.
- `%d` — заменяется днём месяца в виде десятичного числа (01-31).
- `%E* %O*` — расширения локали POSIX. Последовательности `%Ec %EC %Ex %EX %Ey %EY %Od %Oe %OH %OI %Om %OM %OS %Ou %OU %OV %Ow %OW %Oy` должны выводить альтернативные представления.

  Кроме того, альтернативные названия месяцев представляет код формата `%OB` (используется отдельно, без упоминания дня).
- `%e` — заменяется днём в виде десятичного числа (1-31); одиночная цифра предваряется пробелом.
- `%F` — равнозначно `%Y-%m-%d`.
- `%G` — заменяется годом в виде десятичного числа (со столетием). При этом годом считается тот, что содержит наибольшую часть недели (дни недели начинаются с понедельника).
- `%g` — заменяется тем же годом, что и `%G`, но в виде десятичного числа без столетия (00-99).
- `%H` — заменяется часами (в 24-часовом формате) в виде десятичного числа (00-23).
- `%h` — равнозначно `%b`.
- `%I` — заменяется часами (в 12-часовом формате) в виде десятичного числа (01-12).
- `%j` — заменяется днём года в виде десятичного числа (001-366).
- `%k` — заменяется часами (в 24-часовом формате) в виде десятичного числа (0-23); одиночная цифра предваряется пробелом.
- `%l` — заменяется часами (в 12-часовом формате) в виде десятичного числа (1-12); одиночная цифра предваряется пробелом.
- `%M` — заменяется минутами в виде десятичного числа (00-59).
- `%m` — заменяется номером месяца в виде десятичного числа (01-12).
- `%n` — заменяется символом новой строки.
- `%O*` — равнозначно `%E*`.
- `%p` — заменяется локализованным представлением «до полудня» или «после полудня» в зависимости от времени.
- `%R` — равнозначно `%H:%M`.
- `%r` — равнозначно `%I:%M:%S %p`.
- `%S` — заменяется секундами в виде десятичного числа (00-60).
- `%s` — заменяется числом секунд с начала эпохи, по мировому времени (UTC).
- `%T` — равнозначно `%H:%M:%S`
- `%t` — заменяется символом табуляции.
- `%U` — заменяется номером недели в году (первым днём недели считается воскресенье) в виде десятичного числа (00-53).

- `%u` — заменяется номером дня недели (первым днём недели считается понедельник) в виде десятичного числа (1-7).
- `%V` — заменяется номером недели в году (первым днём недели считается понедельник) в виде десятичного числа (01-53). Если к недели, включающей 1 января, относятся 4 или больше дней нового года, она считается неделей с номером 1; в противном случае это последняя неделя предыдущего года, а неделей под номером 1 будет следующая.
- `%v` — равнозначно `%e-%b-%Y`.
- `%W` — заменяется номером недели в году (первым днём недели считается понедельник) в виде десятичного числа (00-53).
- `%w` — заменяется номером дня недели (первым днём недели считается воскресенье) в виде десятичного числа (0-6).
- `%X` — заменяется локализованным представлением времени.
- `%x` — заменяется локализованным представлением даты.
- `%Y` — заменяется годом со столетием в виде десятичного числа.
- `%y` — заменяется годом без столетия в виде десятичного числа (00-99).
- `%Z` — заменяется названием часового пояса.
- `%z` — заменяется смещением часового пояса от UTC; ведущий знак плюс обозначает смещение к востоку от UTC, а знак минус — к западу, часы и минуты задаются парами цифр без разделителя между ними (эта форма установлена для даты в RFC 822).
- `%+` — заменяется локализованным представлением даты и времени.
- `%-*` — расширение GNU libc. Отключает дополнение чисел по ширине при выводе.
- $_* — расширение GNU libc. Явно включает дополнение пробелами.
- `%0*` — расширение GNU libc. Явно включает дополнение нулями.
- `%%` — заменяется символом `%`.

`PGTYPEStimestamp_sub`

Вычитает одно значение времени из другого и сохраняет результат в переменной типа interval.

```
int PGTYPEStimestamp_sub(timestamp *ts1, timestamp *ts2, interval *iv);
```

Эта функция вычитает значение типа timestamp, на которое указывает `ts2`, из значения timestamp, на которое указывает `ts1`, и сохраняет результат в переменной типа interval, на которую указывает `iv`.

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

`PGTYPEStimestamp_defmt_asc`

Разбирает значение типа timestamp из текстового представления с заданной маской формата.

```
int PGTYPEStimestamp_defmt_asc(char *str, char *fmt, timestamp *d);
```

Эта функция получает текстовое представление даты/времени в переменной `str`, а также маску формата для разбора в переменной `fmt`. Результат будет сохранён в переменной, на которую указывает `d`.

Если вместо маски формата `fmt` передаётся NULL, эта функция переходит к стандартной маске форматирования, а именно: `%Y-%m-%d %H:%M:%S`.

Данная функция является обратной к функции `PGTYPEStimestamp_fmt_asc`. Обратитесь к её документации, чтобы узнать о возможных вариантах маски формата.

`PGTYPEStimestamp_add_interval`

Добавляет переменную типа interval к переменной типа timestamp.

```
int PGTYPEStimestamp_add_interval(timestamp *tin, interval *span, timestamp *tout);
```

Эта функция получает указатель на переменную `tin` типа timestamp и указатель на переменную `span` типа interval. Она добавляет временной интервал к значению даты/времени и сохраняет полученную дату/время в переменной типа timestamp, на которую указывает `tout`.

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

`PGTYPEStimestamp_sub_interval`

Вычитает переменную типа interval из переменной типа timestamp.

```
int PGTYPEStimestamp_sub_interval(timestamp *tin, interval *span, timestamp *tout);
```

Эта функция вычитает значение типа interval, на которое указывает `span`, из значения типа timestamp, на которое указывает `tin`, и сохраняет результат в переменной, на которую указывает `tout`.

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

33.6.5. Тип interval

Тип interval, реализованный в C, позволяет программам работать с данными типа interval в SQL. Соответствующий тип сервера Postgres Pro описан в Разделе 8.5.

Для работы с типом interval можно использовать следующие функции:

`PGTYPESinterval_new`

Возвращает указатель на новую переменную interval, размещённую в памяти.

```
interval *PGTYPESinterval_new(void);
```

`PGTYPESinterval_free`

Освобождает место, занимаемое ранее размещённой в памяти переменной типа interval.

```
void PGTYPESinterval_free(interval *intvl);
```

`PGTYPESinterval_from_asc`

Разбирает значение типа interval из его текстового представления.

```
interval *PGTYPESinterval_from_asc(char *str, char **endptr);
```

Эта функция разбирает входную строку `str` и возвращает указатель на размещённую в памяти переменную типа interval. На данный момент ECPG всегда разбирает строку до конца, так что эта функция не может вернуть адрес первого недопустимого символа в `*endptr`. Поэтому в `endptr` свободно можно передать NULL.

`PGTYPESinterval_to_asc`

Преобразует переменную типа interval в текстовое представление.

```
char *PGTYPESinterval_to_asc(interval *span);
```

Эта функция преобразует переменную типа interval, на которую указывает `span`, в строку C char*. Её вывод выглядит примерно так: `@ 1 day 12 hours 59 mins 10 secs`. Результат необходимо освободить функцией `PGTYPESchar_free()`.

`PGTYPESinterval_copy`

Копирует переменную типа interval.

```
int PGTYPESinterval_copy(interval *intvlsrc, interval *intvldest);
```

Эта функция копирует переменную типа interval, на которую указывает `intvlsrc`, в переменную, на которую указывает `intvldest`. Заметьте, что для целевой переменной необходимо предварительно выделить память.

33.6.6. Тип decimal

Тип decimal похож на тип numeric, однако его максимальная точность ограничена 30 значащими цифрами. В отличие от типа numeric, который можно создать только в области кучи, тип decimal можно создать и в стеке, и в области кучи (посредством функций `PGTYPESdecimal_new` и `PGTYPESdecimal_free`). Для работы с типом decimal есть много других функций, подключаемых в режиме совместимости с Informix, описанном в Разделе 33.15.

Для работы с типом decimal можно использовать следующие функции (содержащиеся не в библиотеке `libcompat`).

`PGTYPESdecimal_new`

Запрашивает указатель на новую переменную decimal, размещённую в памяти.

```
decimal *PGTYPESdecimal_new(void);
```

`PGTYPESdecimal_free`

Освобождает переменную типа decimal, высвобождая всю её память.

```
void PGTYPESdecimal_free(decimal *var);
```

33.6.7. Значения errno, которые устанавливает pgtypeslib

`PGTYPES_NUM_BAD_NUMERIC`

Аргумент должен содержать переменную типа numeric (либо указывать на переменную типа numeric), но представление этого типа в памяти оказалось некорректным.

`PGTYPES_NUM_OVERFLOW`

Произошло переполнение. Так как тип numeric может принимать значения практически любой точности, при преобразовании этого типа в другие типы возможно переполнение.

`PGTYPES_NUM_UNDERFLOW`

Произошло антипереполнение. Так как тип numeric может принимать значения практически любой точности, при преобразовании переменной этого типа в другие типы возможно антипереполнение.

`PGTYPES_NUM_DIVIDE_ZERO`

Имела место попытка деления на ноль.

`PGTYPES_DATE_BAD_DATE`

Функции `PGTYPESdate_from_asc` передана некорректная строка даты.

`PGTYPES_DATE_ERR_EARGS`

Функции `PGTYPESdate_defmt_asc` переданы некорректные аргументы.

`PGTYPES_DATE_ERR_ENOSHORTDATE`

В строке, переданной функции `PGTYPESdate_defmt_asc`, оказался неправильный компонент даты.

`PGTYPES_INTVL_BAD_INTERVAL`

Функции `PGTYPESinterval_from_asc` передана некорректная строка, задающая интервал, либо функции `PGTYPESinterval_to_asc` передано некорректное значение интервала.

`PGTYPES_DATE_ERR_ENOTDMY`

Обнаружено несоответствие при выводе компонентов день/месяц/год в функции `PGTYPESdate_defmt_asc`.

PGTYPES_DATE_BAD_DAY

Функция PGTYPESdate_defmt_asc обнаружила некорректное значение дня месяца.

PGTYPES_DATE_BAD_MONTH

Функция PGTYPESdate_defmt_asc обнаружила некорректное значение месяца.

PGTYPES_TS_BAD_TIMESTAMP

Функции PGTYPEStimestamp_from_asc передана некорректная строка даты/времени, либо функции PGTYPEStimestamp_to_asc передано некорректное значение типа timestamp.

PGTYPES_TS_ERR_EINFTIME

Значение типа timestamp, представляющее бесконечность, получено в недопустимом контексте.

33.6.8. Специальные константы pgtypeslib

PGTYPESInvalidTimestamp

Значение типа timestamp, представляющее недопустимое время. Это значение возвращает функция PGTYPEStimestamp_from_asc при ошибке разбора. Заметьте, что вследствие особенности внутреннего представления типа timestamp, значение PGTYPESInvalidTimestamp в то же время представляет корректное время (1899-12-31 23:59:59). Поэтому для выявления ошибок необходимо, чтобы приложение не только сравнивало результат функции с PGTYPESInvalidTimestamp, но и проверяло условие errno != 0 после каждого вызова PGTYPEStimestamp_from_asc.

33.7. Использование областей дескрипторов

Области дескрипторов SQL дают возможности для более сложной обработки результатов операторов SELECT, FETCH и DESCRIBE. Область дескриптора SQL объединяет в одной структуре данные одной строки и элементы метаданных. Эти метаданные особенно полезны при выполнении динамических SQL-операторов, когда характер результирующих столбцов может быть неизвестен заранее. Postgres Pro предлагает два подхода к использованию областей дескрипторов: именованные области SQL-дескрипторов и области SQLDA в структурах C.

33.7.1. Именованные области SQL-дескрипторов

Именованная область SQL-дескриптора состоит из заголовка, содержащего сведения обо всём дескрипторе, и одного или нескольких дескрипторов элементов, которые по сути описывают отдельные столбцы в строке результата.

Прежде чем вы сможете использовать область SQL-дескриптора, её нужно выделить:

```
EXEC SQL ALLOCATE DESCRIPTOR идентификатор;
```

Заданный идентификатор играет роль «имени переменной» области дескриптора. Когда дескриптор оказывается ненужным, его следует освободить:

```
EXEC SQL DEALLOCATE DESCRIPTOR идентификатор;
```

Чтобы воспользоваться областью дескриптора, её нужно указать в качестве целевого объекта в предложении INTO, вместо перечисления переменных среды:

```
EXEC SQL FETCH NEXT FROM mycursor INTO SQL DESCRIPTOR mydesc;
```

Если набор результатов пуст, в области дескриптора будут тем не менее содержаться метаданные из запроса, то есть имена полей.

Получить метаданные набора результатов для ещё не выполненных подготовленных запросов можно, воспользовавшись оператором DESCRIBE:

```
EXEC SQL BEGIN DECLARE SECTION;
```

```
char *sql_stmt = "SELECT * FROM table1";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
```

До PostgreSQL версии 9.0 ключевое слово `SQL` было необязательным, так что предложения `DESCRIPTOR` и `SQL DESCRIPTOR` создавали именованные области SQL-дескрипторов. Теперь оно стало обязательным; без слова `SQL` создаются области SQLDA, см. Подраздел 33.7.2.

В операторах `DESCRIBE` и `FETCH` ключевые слова `INTO` и `USING` действуют примерно одинаково: они указывают вывести набор результатов и метаданные в область дескриптора.

Возникает вопрос: а как же получить данные из области дескриптора? Область дескриптора можно представить как структуру с именованными полями. Чтобы получить значение поля из заголовка и сохранить его в переменной среды C, нужно выполнить команду:

```
EXEC SQL GET DESCRIPTOR имя :переменная_среды = поле;
```

В настоящее время определено только одно поле заголовка: *`COUNT`*, которое говорит, сколько областей дескрипторов элементов существует (то есть, сколько столбцов содержится в результате). Переменная среды C должна иметь целочисленный тип. Чтобы получить поле из области дескриптора элемента, нужно выполнить команду:

```
EXEC SQL GET DESCRIPTOR имя VALUE номер :переменная_среды = поле;
```

В качестве *`num`* можно задать обычное целое или переменную среды C, содержащую целое число. Допустимые поля:

`CARDINALITY` (integer)

число строк в наборе результатов

`DATA`

собственно элемент данных (тип данных поля зависит от запроса)

`DATETIME_INTERVAL_CODE` (целое)

Когда `TYPE` равно `9`, `DATETIME_INTERVAL_CODE` содержит значение `1` для `DATE`, `2` для `TIME`, `3` для `TIMESTAMP`, `4` для `TIME WITH TIME ZONE`, либо `5` для `TIMESTAMP WITH TIME ZONE`.

`DATETIME_INTERVAL_PRECISION` (целое)

не реализовано

`INDICATOR` (целое)

индикатор (отмечающий значение NULL или усечение значения)

`KEY_MEMBER` (целое)

не реализовано

`LENGTH` (целое)

длина данных в символах

`NAME` (строка)

имя столбца

`NULLABLE` (целое)

не реализовано

`OCTET_LENGTH` (целое)

длина символьного представления данных в байтах

`PRECISION` (целое)

точность (для типа `numeric`)

`RETURNED_LENGTH` (целое)

длина данных в символах

`RETURNED_OCTET_LENGTH` (целое)

длина символьного представления данных в байтах

`SCALE` (целое)

масштаб (для типа `numeric`)

`TYPE` (целое)

числовой код типа данных столбца

В операторах `EXECUTE`, `DECLARE` и `OPEN` ключевые слова `INTO` и `USING` действуют по-разному. Область дескриптора также можно сформировать вручную, чтобы передать входные параметры запросу или курсору, а команда `USING SQL DESCRIPTOR` *имя* даёт возможность передать входные аргументы параметризованному запросу. Оператор, формирующий именованную область SQL-дескриптора, выглядит так:

```
EXEC SQL SET DESCRIPTOR имя VALUE номер поле = :переменная_среды;
```

Postgres Pro поддерживает выборку сразу нескольких записей в одном операторе `FETCH` и может сохранить их данные в переменной среды C, если эта переменная — массив. Например:

```
EXEC SQL BEGIN DECLARE SECTION;
int id[5];
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH 5 FROM mycursor INTO SQL DESCRIPTOR mydesc;

EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :id = DATA;
```

33.7.2. Области дескрипторов SQLDA

Область дескриптора SQLDA представляет собой структуру языка C, в которую можно получить набор результатов и метаданные запроса. Одна такая структура содержит одну запись из набора данных.

```
EXEC SQL include sqlda.h;
sqlda_t         *mysqlda;

EXEC SQL FETCH 3 FROM mycursor INTO DESCRIPTOR mysqlda;
```

Заметьте, что ключевое слово `SQL` в этом случае опускается. Замечания относительно применения ключевых слов `INTO` и `USING` в Подразделе 33.7.1 применимы и здесь, с дополнением. В операторе `DESCRIBE` можно полностью опустить ключевое слово `DESCRIPTOR`, если присутствует ключевое слово `INTO`:

```
EXEC SQL DESCRIBE prepared_statement INTO mysqlda;
```

Общая схема использования SQLDA выглядит так:

1. Подготовить запрос и объявить курсор для него.
2. Объявить SQLDA для строк результата.
3. Объявить SQLDA для входных параметров и инициализировать их (выделить память, задать параметры).
4. Открыть курсор с входной SQLDA.

5. Выбрать строки из курсора и сохранить их в выходной SQLDA.
6. Прочитать значения из выходной SQLDA в переменные среды (и преобразовать при необходимости).
7. Закрыть курсор.
8. Освободить область памяти, выделенную для входной SQLDA.

33.7.2.1. Структура данных SQLDA

Для SQLDA используются три типа данных: `sqlda_t`, `sqlvar_t` и `struct sqlname`.

> **Подсказка**
>
> Структура данных SQLDA в Postgres Pro подобна той, что используется в IBM DB2 Universal Database, так что часть технической информации по SQLDA в DB2 может быть полезна и для понимания устройства SQLDA в Postgres Pro.

33.7.2.1.1. Структура sqlda_t

Тип структуры `sqlda_t` представляет тип собственно SQLDA. Эта структура описывает одну запись. Две или более структур `sqlda_t` могут объединяться в связанный список по указателям в поле `desc_next`, и таким образом образовывать упорядоченный набор строк. Поэтому, когда выбираются две или более строк, приложение может прочитать их, проследуя по указателям `desc_next` во всех узлах `sqlda_t`.

Тип `sqlda_t` определяется так:

```
struct sqlda_struct
{
    char            sqldaid[8];
    long            sqldabc;
    short           sqln;
    short           sqld;
    struct sqlda_struct *desc_next;
    struct sqlvar_struct sqlvar[1];
};

typedef struct sqlda_struct sqlda_t;
```

Его поля имеют следующее назначение:

`sqldaid`

Содержит строковую константу `"SQLDA "`.

`sqldabc`

Содержит размер выделенного пространства в байтах.

`sqln`

Содержит число входных параметров для параметризованного запроса, когда передаётся в операторы `OPEN`, `DECLARE` или `EXECUTE` с ключевым словом `USING`. В структуре, выводимой операторами `SELECT`, `EXECUTE` или `FETCH`, данное значение совпадает с `sqld`.

`sqld`

Содержит число полей в наборе результатов.

`desc_next`

Если запрос выдаёт несколько записей, возвращается несколько связанных структур SQLDA, а `desc_next` содержит указатель на следующую запись в списке.

`sqlvar`

Это массив столбцов в наборе результатов.

33.7.2.1.2. Структура sqlvar_t

Тип структуры `sqlvar_t` содержит значение столбца и метаданные, в частности, тип и длину. Эта структура определяется так:

```
struct sqlvar_struct
{
    short          sqltype;
    short          sqllen;
    char          *sqldata;
    short         *sqlind;
    struct sqlname sqlname;
};

typedef struct sqlvar_struct sqlvar_t;
```

Её поля имеют следующее назначение:

`sqltype`

Содержит идентификатор типа данного поля. Возможные значения перечислены в `enum ECPGttype` в `ecpgtype.h`.

`sqllen`

Содержит двоичную длину поля, например 4 байта для `ECPGt_int`.

`sqldata`

Указывает на данные. Формат данных описан в Подразделе 33.4.4.

`sqlind`

Указывает на индикатор NULL. 0 соответствует значению не NULL, -1 — NULL.

`sqlname`

Имя поля.

33.7.2.1.3. Структура struct sqlname

Структура `struct sqlname` содержит имя столбца. Она включена в `sqlvar_t` в качестве члена. Эта структура определена так:

```
#define NAMEDATALEN 64

struct sqlname
{
        short           length;
        char            data[NAMEDATALEN];
};
```

Её поля имеют следующее назначение:

`length`

Содержит длину имени поля.

`data`

Содержит собственно имя поля.

33.7.2.2. Получение набора результатов с применением SQLDA

Чтобы получить набор результатов запроса через SQLDA, нужно проделать примерно следующее:

1. Объявить структуру `sqlda_t` для получения набора результатов.
2. Выполнить команды `FETCH`/`EXECUTE`/`DESCRIBE` для обработки запроса с указанной SQLDA.
3. Определить число записей в наборе результатов, прочитав `sqln`, член структуры `sqlda_t`.
4. Получить значения каждого столбца из элементов `sqlvar[0]`, `sqlvar[1]` и т. д., составляющих массив, включённый в структуру `sqlda_t`.
5. Перейти к следующей строке (структуре `sqlda_t`) по указателю `desc_next`, члену структуры `sqlda_t`.
6. При необходимости повторить эти действия.

Далее показывается, как получить набор результатов через SQLDA.

Сначала объявите структуру `sqlda_t`, в которую будет помещён набор результатов.

```
sqlda_t *sqlda1;
```

Затем укажите эту SQLDA в команде. В данном примере это команда `FETCH`.

```
EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
```

Обработайте все строки в цикле с переходом по связанному списку.

```
sqlda_t *cur_sqlda;

for (cur_sqlda = sqlda1;
     cur_sqlda != NULL;
     cur_sqlda = cur_sqlda->desc_next)
{
    ...
}
```

Внутри этого цикла реализуйте ещё один цикл чтения данных каждого столбца (структуры `sqlvar_t`) в строке.

```
for (i = 0; i < cur_sqlda->sqld; i++)
{
    sqlvar_t v = cur_sqlda->sqlvar[i];
    char *sqldata = v.sqldata;
    short sqllen  = v.sqllen;
    ...
}
```

Чтобы получить значение столбца, прочитайте значение поля `sqltype`, члена структуры `sqlvar_t`. Затем выберите подходящий способ, в зависимости от типа столбца, копирования данных из поля `sqlvar` в переменную среды C.

```
char var_buf[1024];

switch (v.sqltype)
{
    case ECPGt_char:
        memset(&var_buf, 0, sizeof(var_buf));
        memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqllen ? sizeof(var_buf) - 1 :
 sqllen));
        break;

    case ECPGt_int: /* integer */
        memcpy(&intval, sqldata, sqllen);
        snprintf(var_buf, sizeof(var_buf), "%d", intval);
        break;
```

```
    ...
}
```

33.7.2.3. Передача значений параметров через SQLDA

Чтобы передать параметры подготовленному запросу через SQLDA, нужно проделать примерно следующее:

1. Создать подготовленный запрос (подготовленный оператор)
2. Объявить структуру sqlda_t в качестве входной SQLDA.
3. Выделить область памяти (структуру sqlda_t) для входной SQLDA.
4. Установить (скопировать) входные значения в выделенной памяти.
5. Открыть курсор, указав входную SQLDA.

Рассмотрим это на примере.

Сначала создайте подготовленный оператор.

```
EXEC SQL BEGIN DECLARE SECTION;
char query[1024] = "SELECT d.oid, * FROM pg_database d, pg_stat_database s WHERE d.oid
 = s.datid AND (d.datname = ? OR d.oid = ?)";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE stmt1 FROM :query;
```

Затем выделите память для SQLDA и установите число входных параметров в поле `sqln`, члене структуры `sqlda_t`. Когда для подготовленного запроса требуются два или более входных параметров, приложение должно выделить дополнительное место в памяти, размер которого вычисляется как (число параметров - 1) * sizeof(sqlvar_t). В показанном здесь примере выделяется место для двух параметров.

```
sqlda_t *sqlda2;

sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));

sqlda2->sqln = 2; /* число входных переменных */
```

Выделив память, сохраните значения параметров в массиве `sqlvar[]`. (Этот же массив используется для значений столбцов, когда SQLDA получает набор результатов.) В данном примере передаются два параметра: `"postgres"` (строкового типа) и `1` (целочисленного типа).

```
sqlda2->sqlvar[0].sqltype = ECPGt_char;
sqlda2->sqlvar[0].sqldata = "postgres";
sqlda2->sqlvar[0].sqllen  = 8;

int intval = 1;
sqlda2->sqlvar[1].sqltype = ECPGt_int;
sqlda2->sqlvar[1].sqldata = (char *) &intval;
sqlda2->sqlvar[1].sqllen  = sizeof(intval);
```

Откройте курсор с указанием ранее созданной SQLDA, чтобы входные параметры были переданы подготовленному оператору.

```
EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;
```

Наконец, закончив использование входных SQLDA, необходимо явно освободить выделенную для них память, в отличие от SQLDA, получающих результаты запросов.

```
free(sqlda2);
```

33.7.2.4. Пример приложения, использующего SQLDA

Представленный здесь пример программы показывает, как выбрать из системных каталогов статистику доступа к базам данных, определённых входными параметрами.

Это приложение соединяет записи двух системных таблиц, pg_database и pg_stat_database по OID базы данных, и также выбирает и показывает статистику, принимая два входных параметра (база данных `postgres` и OID `1`).

Сначала создайте SQLDA для ввода параметров и SQLDA для вывода результатов.

```
EXEC SQL include sqlda.h;

sqlda_t *sqlda1; /* выходной дескриптор */
sqlda_t *sqlda2; /* входной дескриптор */
```

Затем подключитесь к базе данных, подготовьте оператор и объявите курсор для подготовленного оператора.

```
int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d, pg_stat_database s WHERE
 d.oid=s.datid AND ( d.datname=? OR d.oid=? )";
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

Затем запишите некоторые значения параметров во входную SQLDA. Выделите память для входной SQLDA и установите количество параметров в `sqln`. Запишите тип, значение и длину значения в поля `sqltype`, `sqldata` и `sqllen` структуры `sqlvar`.

```
    /* Создать структуру SQLDA для входных параметров. */
    sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
    memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));
    sqlda2->sqln = 2; /* число входных переменных */

    sqlda2->sqlvar[0].sqltype = ECPGt_char;
    sqlda2->sqlvar[0].sqldata = "postgres";
    sqlda2->sqlvar[0].sqllen  = 8;

    intval = 1;
    sqlda2->sqlvar[1].sqltype = ECPGt_int;
    sqlda2->sqlvar[1].sqldata = (char *)&intval;
    sqlda2->sqlvar[1].sqllen  = sizeof(intval);
```

Подготовив входную SQLDA, откройте курсор с ней.

```
    /* Открыть курсор с входными параметрами. */
    EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;
```

Выберите строки из открытого курсора в выходную SQLDA. (Обычно, чтобы выбрать все строки в наборе результатов, нужно повторять `FETCH` в цикле.)

```
    while (1)
    {
        sqlda_t *cur_sqlda;
```

```
        /* Назначить дескриптор курсору */
        EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
```

Затем прочитайте выбранные записи из SQLDA, следуя по связанному списку структуры `sqlda_t`.

```
    for (cur_sqlda = sqlda1 ;
         cur_sqlda != NULL ;
         cur_sqlda = cur_sqlda->desc_next)
    {
        ...
```

Прочитайте все столбцы первой записи. Количество столбцов хранится в поле `sqld`, а данные первого столбца в `sqlvar[0]`, оба эти поля — члены структуры `sqlda_t`.

```
        /* Вывести каждый столбец в строке. */
        for (i = 0; i < sqlda1->sqld; i++)
        {
            sqlvar_t v = sqlda1->sqlvar[i];
            char *sqldata = v.sqldata;
            short sqllen  = v.sqllen;

            strncpy(name_buf, v.sqlname.data, v.sqlname.length);
            name_buf[v.sqlname.length] = '\0';
```

Теперь данные столбцов сохранены в переменной `v`. Скопируйте все элементы данных в переменные среды, определив тип столбца по полю `v.sqltype`.

```
            switch (v.sqltype) {
                int intval;
                double doubleval;
                unsigned long long int longlongval;

                case ECPGt_char:
                    memset(&var_buf, 0, sizeof(var_buf));
                    memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqllen ?
 sizeof(var_buf)-1 : sqllen));
                    break;

                case ECPGt_int: /* integer */
                    memcpy(&intval, sqldata, sqllen);
                    snprintf(var_buf, sizeof(var_buf), "%d", intval);
                    break;

                ...

                default:
                    ...
            }

            printf("%s = %s (type: %d)\n", name_buf, var_buf, v.sqltype);
        }
```

Закончив обработку всех записей, закройте курсор и отключитесь от базы данных.

```
    EXEC SQL CLOSE cur1;
    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT ALL;
```

Вся программа показана в Примере 33.1.

Пример 33.1. Пример программы на базе SQLDA

```
#include <stdlib.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

EXEC SQL include sqlda.h;

sqlda_t *sqlda1; /* дескриптор для выходных данных */
sqlda_t *sqlda2; /* дескриптор для входных данных */

EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d, pg_stat_database s WHERE
 d.oid=s.datid AND ( d.datname=? OR d.oid=? )";

    int intval;
    unsigned long long int longlongval;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO uptimedb AS con1 USER uptime;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;

    /* Создать структуру SQLDA для входных параметров */
    sqlda2 = (sqlda_t *)malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
    memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));
    sqlda2->sqln = 2; /* число входных переменных */

    sqlda2->sqlvar[0].sqltype = ECPGt_char;
    sqlda2->sqlvar[0].sqldata = "postgres";
    sqlda2->sqlvar[0].sqllen  = 8;

    intval = 1;
    sqlda2->sqlvar[1].sqltype = ECPGt_int;
    sqlda2->sqlvar[1].sqldata = (char *) &intval;
    sqlda2->sqlvar[1].sqllen  = sizeof(intval);

    /* Открыть курсор с входными параметрами. */
    EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;

    while (1)
    {
        sqlda_t *cur_sqlda;

        /* Присвоить дескриптор курсору */
        EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;

        for (cur_sqlda = sqlda1 ;
```

```
             cur_sqlda != NULL ;
             cur_sqlda = cur_sqlda->desc_next)
        {
            int i;
            char name_buf[1024];
            char var_buf[1024];

            /* Напечатать каждый столбец в строке. */
            for (i=0 ; i<cur_sqlda->sqld ; i++)
            {
                sqlvar_t v = cur_sqlda->sqlvar[i];
                char *sqldata = v.sqldata;
                short sqllen  = v.sqllen;

                strncpy(name_buf, v.sqlname.data, v.sqlname.length);
                name_buf[v.sqlname.length] = '\0';

                switch (v.sqltype)
                {
                    case ECPGt_char:
                        memset(&var_buf, 0, sizeof(var_buf));
                        memcpy(&var_buf, sqldata, (sizeof(var_buf)<=sqllen ?
sizeof(var_buf)-1 : sqllen) );
                        break;

                    case ECPGt_int: /* integer */
                        memcpy(&intval, sqldata, sqllen);
                        snprintf(var_buf, sizeof(var_buf), "%d", intval);
                        break;

                    case ECPGt_long_long: /* bigint */
                        memcpy(&longlongval, sqldata, sqllen);
                        snprintf(var_buf, sizeof(var_buf), "%lld", longlongval);
                        break;

                    default:
                    {
                        int i;
                        memset(var_buf, 0, sizeof(var_buf));
                        for (i = 0; i < sqllen; i++)
                        {
                            char tmpbuf[16];
                            snprintf(tmpbuf, sizeof(tmpbuf), "%02x ", (unsigned char)
sqldata[i]);
                            strncat(var_buf, tmpbuf, sizeof(var_buf));
                        }
                    }
                        break;
                }

                printf("%s = %s (type: %d)\n", name_buf, var_buf, v.sqltype);
            }

            printf("\n");
        }
    }

    EXEC SQL CLOSE cur1;
```

```
    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

Вывод этой программы должен быть примерно таким (некоторые числа будут меняться).

```
oid = 1 (type: 1)
datname = template1 (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = t (type: 1)
datallowconn = t (type: 1)
datconnlimit = -1 (type: 5)
datlastsysoid = 11510 (type: 1)
datfrozenxid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig =  (type: 1)
datacl = {=c/uptime,uptime=CTc/uptime} (type: 1)
datid = 1 (type: 1)
datname = template1 (type: 1)
numbackends = 0 (type: 5)
xact_commit = 113606 (type: 9)
xact_rollback = 0 (type: 9)
blks_read = 130 (type: 9)
blks_hit = 7341714 (type: 9)
tup_returned = 38262679 (type: 9)
tup_fetched = 1836281 (type: 9)
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)

oid = 11511 (type: 1)
datname = postgres (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = f (type: 1)
datallowconn = t (type: 1)
datconnlimit = -1 (type: 5)
datlastsysoid = 11510 (type: 1)
datfrozenxid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig =  (type: 1)
datacl =  (type: 1)
datid = 11511 (type: 1)
datname = postgres (type: 1)
numbackends = 0 (type: 5)
xact_commit = 221069 (type: 9)
xact_rollback = 18 (type: 9)
blks_read = 1176 (type: 9)
blks_hit = 13943750 (type: 9)
tup_returned = 77410091 (type: 9)
tup_fetched = 3253694 (type: 9)
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)
```

33.8. Обработка ошибок

В этом разделе описывается, как можно обрабатывать исключительные условия и предупреждения в программе со встраиваемым SQL. Для этого предназначены два средства, которые могут дополнять друг друга.

- Можно настроить функции-обработчики для обработки предупреждений и ошибок, воспользовавшись командой `WHENEVER`.
- Подробную информацию об ошибке или предупреждении можно получить через переменную `sqlca`.

33.8.1. Установка обработчиков

Один простой метод перехвата ошибок и предупреждений заключается в назначении определённого действия, которое будет выполняться при некотором условии. В общем виде:

```
EXEC SQL WHENEVER условие действие;
```

Здесь *условие* может быть следующим:

`SQLERROR`

Указанное действие вызывается, когда при выполнении SQL-оператора происходит ошибка.

`SQLWARNING`

Указанное действие вызывается, когда при выполнении SQL-оператора выдаётся предупреждение.

`NOT FOUND`

Указанное действие вызывается, когда SQL-оператор получает или обрабатывает ноль строк. (Это обстоятельство не считается ошибкой, но бывает полезно отследить его.)

действие может быть следующим:

`CONTINUE`

Это фактически означает, что условие игнорируется. Это поведение по умолчанию.

`GOTO` *метка*
`GO TO` *метка*

Перейти к указанной метке (используя оператор `goto` языка C).

`SQLPRINT`

Вывести сообщение в устройство стандартного вывода. Это полезно для простых программ или при разработке прототипов. Содержание этого сообщения не настраивается.

`STOP`

Вызвать `exit(1)`, что приведёт к завершению программы.

`DO BREAK`

Выполнить оператор `break` языка C. Этот вариант следует использовать только в циклах или операторах `switch`.

`CALL` *имя* (*аргументы*)
`DO` *имя* (*аргументы*)

Вызвать указанные функции C с заданными аргументами.

В стандарте SQL описаны только действия `CONTINUE` и `GOTO` (и `GO TO`).

Ниже показан простой пример использования этих команд. Эта конструкция выводит простое сообщение при выдаче предупреждения и прерывает программу в случае ошибки:

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

Оператор `EXEC SQL WHENEVER` является директивой препроцессора SQL, а не оператором языка C. Устанавливаемое им действие при ошибках или предупреждениях применяется ко всем встраиваемым SQL-операторам ниже точки, где устанавливается обработчик, если только это действие не было изменено после первой команды `EXEC SQL WHENEVER`, и до SQL-оператора, вызвавшего это условие, вне зависимости от хода выполнения программы на C. Поэтому обе следующие программы на C не дадут желаемого эффекта:

```
/*
 * НЕПРАВИЛЬНО
 */
int main(int argc, char *argv[])
{
    ...
    if (verbose) {
        EXEC SQL WHENEVER SQLWARNING SQLPRINT;
    }
    ...
    EXEC SQL SELECT ...;
    ...
}
```

```
/*
 * НЕПРАВИЛЬНО
 */
int main(int argc, char *argv[])
{
    ...
    set_error_handler();
    ...
    EXEC SQL SELECT ...;
    ...
}

static void set_error_handler(void)
{
    EXEC SQL WHENEVER SQLERROR STOP;
}
```

33.8.2. sqlca

Для более гибкой обработки ошибок в интерфейсе встраиваемого SQL представлена глобальная переменная с именем `sqlca` (SQL Communication Area, Область сведений SQL), имеющая следующую структуру:

```
struct
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        int sqlerrml;
        char sqlerrmc[SQLERRMC_LEN];
    } sqlerrm;
    char sqlerrp[8];
```

```
    long sqlerrd[6];
    char sqlwarn[8];
    char sqlstate[5];
} sqlca;
```

(В многопоточной программе каждый поток автоматически получает собственную копию `sqlca`. Это работает подобно стандартной в C глобальной переменной `errno`.)

Структура `sqlca` покрывает и предупреждения, и ошибки. Если в процессе выполнения оператора выдаётся несколько предупреждений или ошибок, `sqlca` будет содержать сведения только о последнем(ей) из них.

Если последний оператор SQL выполняется без ошибки, `sqlca.sqlcode` будет содержать 0, а `sqlca.sqlstate` — `"00000"`. Если выдаётся предупреждение или ошибка, в `sqlca.sqlcode` будет содержаться отрицательное число, а `sqlca.sqlstate` будет отличаться от `"00000"`. Положительное значение `sqlca.sqlcode` устанавливается при нейтральном событии, например, когда последний запрос возвращает ноль строк. Поля `sqlcode` и `sqlstate` представляют две различные схемы кодов ошибок; подробнее они описаны ниже.

Если последний SQL-оператор был успешным, в `sqlca.sqlerrd[1]` содержится OID обработанной строки (если это уместно), а в `sqlca.sqlerrd[2]` количество обработанных или возвращённых строк (если это уместно для команды).

В случае ошибки или предупреждения `sqlca.sqlerrm.sqlerrmc` будет содержать строку, описывающую ошибку. Поле `sqlca.sqlerrm.sqlerrml` содержит длину сообщения об ошибке, которое хранится в `sqlca.sqlerrm.sqlerrmc` (результат функции `strlen()`, который не очень интересен для программиста C). Заметьте, что некоторые сообщения могут не умещаться в массив `sqlerrmc` фиксированного размера; они будут обрезаться.

В случае предупреждения, в `sqlca.sqlwarn[2]` записывается символ `W`. (Во всех других случаях значение будет отличным от `W`.) Символ `W` в `sqlca.sqlwarn[1]` показывает, что значение было обрезано при сохранении в переменной среды C. `W` в `sqlca.sqlwarn[0]` устанавливается, если предупреждение отмечается в каком-либо другом элементе массива.

Поля `sqlcaid`, `sqlabc`, `sqlerrp` и остальные элементы `sqlerrd` и `sqlwarn` в настоящее время не содержат полезной информации.

Структура `sqlca` не определена в стандарте SQL, но реализована в нескольких других СУБД SQL. Принципиально она определяется одинаково, но если вы хотите, чтобы ваши приложения были переносимыми, тщательно изучите различия реализаций.

В следующем примере, демонстрирующем применение `WHENEVER` в сочетании с `sqlca`, выводится содержимое `sqlca` при возникновении ошибки. Это может быть полезно для отладки или в прототипах, пока не реализован более «дружественный пользователю» обработчик ошибок.

```
EXEC SQL WHENEVER SQLERROR CALL print_sqlca();

void
print_sqlca()
{
    fprintf(stderr, "==== sqlca ====\n");
    fprintf(stderr, "sqlcode: %ld\n", sqlca.sqlcode);
    fprintf(stderr, "sqlerrm.sqlerrml: %d\n", sqlca.sqlerrm.sqlerrml);
    fprintf(stderr, "sqlerrm.sqlerrmc: %s\n", sqlca.sqlerrm.sqlerrmc);
    fprintf(stderr, "sqlerrd: %ld %ld %ld %ld %ld %ld\n",
 sqlca.sqlerrd[0],sqlca.sqlerrd[1],sqlca.sqlerrd[2],

 sqlca.sqlerrd[3],sqlca.sqlerrd[4],sqlca.sqlerrd[5]);
    fprintf(stderr, "sqlwarn: %d %d %d %d %d %d %d %d\n", sqlca.sqlwarn[0],
 sqlca.sqlwarn[1], sqlca.sqlwarn[2],
```

```
                                                                sqlca.sqlwarn[3],
 sqlca.sqlwarn[4], sqlca.sqlwarn[5],
                                                                sqlca.sqlwarn[6],
 sqlca.sqlwarn[7]);
    fprintf(stderr, "sqlstate: %5s\n", sqlca.sqlstate);
    fprintf(stderr, "===============\n");
}
```

Результат может выглядеть следующим образом (при ошибке, вызванной опечаткой в имени таблицы):

```
==== sqlca ====
sqlcode: -400
sqlerrm.sqlerrml: 49
sqlerrm.sqlerrmc: relation "pg_databasep" does not exist on line 38
sqlerrd: 0 0 0 0 0 0
sqlwarn: 0 0 0 0 0 0 0 0
sqlstate: 42P01
===============
```

33.8.3. SQLSTATE и SQLCODE

Поля `sqlca.sqlstate` и `sqlca.sqlcode` отражают две различные схемы, представляющие коды ошибок. Обе схемы пришли из стандарта SQL, но схема `SQLCODE` была признана устаревшей в редакции SQL-92 и исключена в последующих редакциях. Поэтому в новых приложениях настоятельно рекомендуется использовать `SQLSTATE`.

`SQLSTATE` задаётся в массиве из пяти символов. Эти пять символов содержат цифры или буквы в верхнем регистре, представляющие коды различных условий ошибок и предупреждений. `SQLSTATE` определяется по иерархической схеме: первые два символа обозначают общий класс условия, а следующие три — подкласс общего условия. Успешное состояние обозначается кодом `00000`. По большей части коды `SQLSTATE` определены в стандарте SQL. Сервер Postgres Pro поддерживает коды ошибок `SQLSTATE` естественным образом; поэтому используя во всех приложениях именно эту схему кодов ошибок, можно добиться высокой степени согласованности. За дальнейшими сведениями обратитесь к Приложению А.

`SQLCODE` — устаревшая схема, в которой коды ошибок представлены просто целыми числами. Значение 0 обозначает успешное завершение, положительное значение — успешное завершение с дополнительной информацией, а отрицательное говорит об ошибке. В стандарте SQL определено только положительное значение +100, показывающее, что последняя команда вернула или затронула ноль строк, но отрицательные значения не определены. Таким образом, с этой схемой нельзя рассчитывать на переносимость и она не имеет иерархической структуры. Исторически сложилось, что процессор встраиваемого SQL для Postgres Pro назначает некоторые определённые значения `SQLCODE` для собственного использования; они перечислены ниже с числовыми значениями и символьными именами. Помните, что эти коды несовместимы с другими реализациями SQL. Поэтому для упрощения перевода приложений на схему `SQLSTATE` вместе с этими кодами перечисляются соответствующие значения `SQLSTATE`. Однако однозначного соответствия один-к-одному или один-ко-многим между этими двумя схемами не существует (на самом деле это соответствие многие-ко-многим), поэтому следует свериться со списком `SQLSTATE` в Приложении А в каждом случае.

`SQLCODE` может принимать следующие значения:

0 (`ECPG_NO_ERROR`)

Показывает, что ошибки нет. (SQLSTATE 00000)

100 (`ECPG_NOT_FOUND`)

Это нейтральное условие, показывающее, что последняя команда вернула или обработала ноль строк, либо курсор достиг конца. (SQLSTATE 02000)

Выбирая данные из курсора в цикле, можно проверять этот код, чтобы понять, когда нужно прервать цикл, следующим образом:

```
while (1)
{
    EXEC SQL FETCH ... ;
    if (sqlca.sqlcode == ECPG_NOT_FOUND)
        break;
}
```

Но `WHENEVER NOT FOUND DO BREAK` внутри по сути делает это же, поэтому такое явное условие обычно ничем не лучше.

-12 (`ECPG_OUT_OF_MEMORY`)

Указывает, что закончилась виртуальная память. Числовое значение определено как `-ENOMEM`. (SQLSTATE YE001)

-200 (`ECPG_UNSUPPORTED`)

Указывает, что препроцессор сгенерировал код, который не понимает библиотека. Возможно, вы используете несовместимые версии препроцессора и библиотеки. (SQLSTATE YE002)

-201 (`ECPG_TOO_MANY_ARGUMENTS`)

Это означает, что в команде было указано больше переменных среды, чем она ожидает. (SQLSTATE 07001 или 07002)

-202 (`ECPG_TOO_FEW_ARGUMENTS`)

Это означает, что в команде было указано меньше переменных среды, чем она ожидает. (SQLSTATE 07001 или 07002)

-203 (`ECPG_TOO_MANY_MATCHES`)

Это означает, что запрос вернул несколько строк, но оператор был подготовлен только для одной строки результата (например, потому что переданные переменные — не массивы). (SQLSTATE 21000)

-204 (`ECPG_INT_FORMAT`)

Переменная среды C типа `int` и данные в базе имеют разные типы, и в этих данных содержится значение, которое нельзя преобразовать в `int`. Для этого преобразования библиотека использует функцию `strtol()`. (SQLSTATE 42804)

-205 (`ECPG_UINT_FORMAT`)

Переменная среды C типа `unsigned int` и данные в базе имеют разные типы, и в этих данных содержится значение, которое нельзя преобразовать в `unsigned int`. Для этого преобразования библиотека использует функцию `strtoul()`. (SQLSTATE 42804)

-206 (`ECPG_FLOAT_FORMAT`)

Переменная среды C типа `float` и данные в базе имеют разные типы, и в этих данных содержится значение, которое нельзя преобразовать во `float`. Для этого преобразования библиотека использует функцию `strtod()`. (SQLSTATE 42804)

-207 (`ECPG_NUMERIC_FORMAT`)

Переменная среды C типа `numeric` и данные в базе имеют разные типы, и в этих данных содержится значение, которое нельзя преобразовать в `numeric`. (SQLSTATE 42804)

-208 (`ECPG_INTERVAL_FORMAT`)

Переменная среды C типа `interval` и данные в базе имеют разные типы, и в этих данных содержится значение, которое нельзя преобразовать в `interval`. (SQLSTATE 42804)

-209 (ECPG_DATE_FORMAT)

Переменная среды C типа date и данные в базе имеют разные типы, и в этих данных содержится значение, которое нельзя преобразовать в date. (SQLSTATE 42804)

-210 (ECPG_TIMESTAMP_FORMAT)

Переменная среды C типа timestamp и данные в базе имеют разные типы, и в этих данных содержится значение, которое нельзя преобразовать в timestamp. (SQLSTATE 42804)

-211 (ECPG_CONVERT_BOOL)

Это означает, что переменная среды C имеет тип bool, а значение в базе данных отличается от 't' или 'f'. (SQLSTATE 42804)

-212 (ECPG_EMPTY)

Серверу Postgres Pro был передан пустой оператор. (Этого обычно не должно происходить в программе со встраиваемым SQL, так что это может указывать на внутреннюю ошибку.) (SQLSTATE YE002)

-213 (ECPG_MISSING_INDICATOR)

Возвращено значение NULL, но переменная-индикатор NULL не задана. (SQLSTATE 22002)

-214 (ECPG_NO_ARRAY)

Там, где требуется массив, была передана обычная переменная. (SQLSTATE 42804)

-215 (ECPG_DATA_NOT_ARRAY)

База данных возвратила обычную переменную там, где требуется значение-массив. (SQLSTATE 42804)

-216 (ECPG_ARRAY_INSERT)

Не удалось вставить значение в массив. (SQLSTATE 42804)

-220 (ECPG_NO_CONN)

Программа попыталась использовать несуществующее подключение. (SQLSTATE 08003)

-221 (ECPG_NOT_CONN)

Программа попыталась использовать подключение, которое существует, но не было открыто. (Это внутренняя ошибка.) (SQLSTATE YE002)

-230 (ECPG_INVALID_STMT)

Оператор, который вы пытаетесь выполнить, не был подготовлен. (SQLSTATE 26000)

-239 (ECPG_INFORMIX_DUPLICATE_KEY)

Ошибка повторяющегося ключа, нарушение ограничения уникальности (режим совместимости с Informix). (SQLSTATE 23505)

-240 (ECPG_UNKNOWN_DESCRIPTOR)

Указанный дескриптор не найден. Оператор, который вы пытаетесь использовать, не был подготовлен. (SQLSTATE 33000)

-241 (ECPG_INVALID_DESCRIPTOR_INDEX)

Указанный индекс дескриптора вне диапазона. (SQLSTATE 07009)

-242 (ECPG_UNKNOWN_DESCRIPTOR_ITEM)

Запрошен неверный элемент дескриптора. (Это внутренняя ошибка.) (SQLSTATE YE002)

-243 (ECPG_VAR_NOT_NUMERIC)

При выполнении динамического оператора база данных возвратила числовое значение, тогда как переменная среды C — не числовая. (SQLSTATE 07006)

-244 (ECPG_VAR_NOT_CHAR)

При выполнении динамического оператора база данных возвратила не числовое значение, тогда как переменная среды C — числовая. (SQLSTATE 07006)

-284 (ECPG_INFORMIX_SUBSELECT_NOT_ONE)

Результат подзапроса представлен не одной строкой (режим совместимости с Informix). (SQLSTATE 21000)

-400 (ECPG_PGSQL)

Ошибка произошла на стороне сервера Postgres Pro. В тексте ошибки содержится сообщение об ошибке от сервера Postgres Pro.

-401 (ECPG_TRANS)

Сервер Postgres Pro сообщает, что клиент не может запускать, фиксировать или отменять транзакцию. (SQLSTATE 08007)

-402 (ECPG_CONNECT)

Попытка подключения к базе данных была неудачной. (SQLSTATE 08001)

-403 (ECPG_DUPLICATE_KEY)

Ошибка повторяющегося ключа, нарушение ограничения уникальности. (SQLSTATE 23505)

-404 (ECPG_SUBSELECT_NOT_ONE)

Результат подзапроса представлен не одной строкой. (SQLSTATE 21000)

-602 (ECPG_WARNING_UNKNOWN_PORTAL)

Указано неверное имя курсора. (SQLSTATE 34000)

-603 (ECPG_WARNING_IN_TRANSACTION)

Транзакция в процессе выполнения. (SQLSTATE 25001)

-604 (ECPG_WARNING_NO_TRANSACTION)

Нет активной (выполняющейся) транзакции. (SQLSTATE 25P01)

-605 (ECPG_WARNING_PORTAL_EXISTS)

Было указано имя существующего курсора. (SQLSTATE 42P03)

33.9. Директивы препроцессора

Препроцессор ecpg поддерживает ряд директив, которые позволяют управлять разбором и обработкой исходных файлов.

33.9.1. Включение файлов

Для включения внешнего файла в программу со встраиваемым SQL, используется конструкция:

```
EXEC SQL INCLUDE имя_файла;
EXEC SQL INCLUDE <имя_файла>;
EXEC SQL INCLUDE "имя_файла";
```

Встретив такую директиву, препроцессор встраиваемого SQL будет искать файл *имя_файла*.h, обрабатывать его и включать в выходной код C. В результате встраиваемые SQL-операторы во включённом таким образом файле будут обработаны корректно.

Препроцессор ecpg будет искать указанный файл в нескольких каталогах в следующем порядке:

- текущий каталог
- /usr/local/include
- каталог включаемых файлов Postgres Pro, определённый во время сборки (например, /usr/local/pgsql/include)
- /usr/include

Но когда используется форма EXEC SQL INCLUDE "*имя_файла*", просматривается только текущий каталог.

В каждом каталоге препроцессор будет сначала искать файл с заданным именем, а если не обнаружит его, попытается найти файл с добавленным расширением .h (если только заданное имя файла уже не содержит это расширение).

Заметьте, что команда EXEC SQL INCLUDE *не* равнозначна включению:

```
#include <имя_файла.h>
```

так как во втором случае включаемый файл не проходит через препроцессор SQL-команд. Естественно, директиву C #include можно по-прежнему применять для включения других заголовочных файлов.

> **Примечание**
>
> Имя включаемого файла чувствительно к регистру, несмотря на то, что остальная команда EXEC SQL INCLUDE подчиняется обычным правилам чувствительности к регистру SQL.

33.9.2. Директивы define и undef

Во встраиваемом SQL есть конструкция, подобная директиве #define, известной в C:

```
EXEC SQL DEFINE имя;
EXEC SQL DEFINE имя значение;
```

Используя её, можно определить имя:

```
EXEC SQL DEFINE HAVE_FEATURE;
```

И также можно определить константы:

```
EXEC SQL DEFINE MYNUMBER 12;
EXEC SQL DEFINE MYSTRING 'abc';
```

Удалить предыдущее определение позволяет команда undef:

```
EXEC SQL UNDEF MYNUMBER;
```

Разумеется, в программе со встраиваемым SQL можно продолжать использовать версии #define и #undef языка C. Отличие состоит в том, когда вычисляются определяемые значения. Когда применяется команда EXEC SQL DEFINE, вычислять определения и подставлять значения будет препроцессор ecpg. Например, если написать:

```
EXEC SQL DEFINE MYNUMBER 12;
...
EXEC SQL UPDATE Tbl SET col = MYNUMBER;
```

подстановку выполнит ecpg и компилятор C никогда не увидит имени или идентификатора MYNUMBER. Заметьте, что с другой стороны #define не подходит для определения константы,

которую вы хотите использовать во встраиваемом SQL, так как препроцессор встраиваемого SQL не сможет увидеть это определение.

33.9.3. Директивы ifdef, ifndef, else, elif и endif

Для условной компиляции блоков кода можно использовать следующие указания:

`EXEC SQL ifdef` *`имя`*`;`

Проверяет *`имя`* и обрабатывает последующие строки, если *`имя`* было определено командой `EXEC SQL define` *`имя`*.

`EXEC SQL ifndef` *`имя`*`;`

Проверяет *`имя`* и обрабатывает последующие строки, если *`имя`* ***не*** было определено командой `EXEC SQL define` *`имя`*.

`EXEC SQL else;`

Начинает альтернативную ветвь условия, открытого командой `EXEC SQL ifdef` *`имя`* или `EXEC SQL ifndef` *`имя`*.

`EXEC SQL elif` *`имя`*`;`

Проверяет *`имя`* и начинает альтернативную ветвь условия, если *`имя`* было определено командой `EXEC SQL define` *`имя`*.

`EXEC SQL endif;`

Завершает ветвь условия.

Пример:

```
EXEC SQL ifndef TZVAR;
EXEC SQL SET TIMEZONE TO 'GMT';
EXEC SQL elif TZNAME;
EXEC SQL SET TIMEZONE TO TZNAME;
EXEC SQL else;
EXEC SQL SET TIMEZONE TO TZVAR;
EXEC SQL endif;
```

33.10. Компиляция программ со встраиваемым SQL

Теперь, когда вы получили представление, как писать программы на C со встраиваемым SQL, вы наверное хотите узнать, как их компилировать. Прежде чем компилировать код C, необходимо пропустить исходный файл через препроцессор встраиваемого SQL в C, который преобразует записанные вами операторы SQL в вызовы специальных функций. После компиляции полученный объектный код нужно скомпоновать со специальной библиотекой, содержащей необходимые функции. Эти функции получают информацию из аргументов, выполняют команды SQL через интерфейс libpq, и помещают результат в аргументы, заданные для вывода.

Программа препроцессора называется `ecpg` и входит в состав обычной инсталляции Postgres Pro. Программам со встраиваемым SQL, как правило, даются имена с расширением `.pgc`. Если вы создали код программы в файле `prog1.pgc`, вы можете обработать его, просто выполнив:

```
ecpg prog1.pgc
```

При этом будет создан файл `prog1.c`. Если имена входных файлов не следуют этому соглашению, имя выходного файла можно задать явно в аргументе `-o`.

Обработанный препроцессором файл можно скомпилировать обычным образом, например, так:

```
cc -c prog1.c
```

В сгенерированные исходные файлы C включаются заголовочные файлы из инсталляции Postgres Pro, поэтому если вы установили Postgres Pro так, что соответствующий каталог не

просматривается по умолчанию, вам придётся добавить указание вида `-I/usr/local/pgsql/include` в командную строку компиляции.

Чтобы скомпоновать программу со встраиваемым SQL, необходимо подключить библиотеку `libecpg` примерно так:

```
cc -o myprog prog1.o prog2.o ... -lecpg
```

Возможно, и для этого понадобится добавить в командную строку указание вида `-L/usr/local/pgsql/lib`.

Чтобы узнать пути вашей инсталляции, можно воспользоваться командой `pg_config`или `pkg-config`(в качестве имени пакета нужно указать `libecpg`).

Если вы организуете процесс сборки большого проекта с применением make, может быть удобно включить в ваши сборочные файлы следующее неявное правило:

```
ECPG = ecpg

%.c: %.pgc
        $(ECPG) $<
```

Полный синтаксис вызова команды `ecpg` приведён в описании ecpg.

Библиотека ecpg по умолчанию потокобезопасна. Однако для компиляции клиентского кода могут потребоваться параметры командной строки для настройки многопоточности.

33.11. Библиотечные функции

Библиотека `libecpg` в основном содержит «скрытые» функции, применяемые для реализации функциональности, выражаемой встраиваемыми командами SQL. Но есть также некоторые функции, которые можно вызывать напрямую. Заметьте, что код, задействующий эти функции, будет непереносимым.

- `ECPGdebug(int` *`вкл`*`, FILE *`*`поток`*`)` с первым аргументом, отличным от нуля, включает вывод отладочных сообщений в заданный *`поток`*. Журнал сообщений, полученный таким образом, будет содержать все операторы SQL с заданными входными переменными и результаты, выданные сервером Postgres Pro. Это может быть очень полезно для поиска ошибок в командах SQL.

  > **Примечание**
  >
  > В Windows, если библиотека ecpg и приложение скомпилированы с разными флагами, эта функция может вызвать крах приложения из-за различий внутреннего представления указателей `FILE`. В частности, флаги многопоточной/однопоточной, выпускаемой/отладочной или статической/динамической сборки должны быть одинаковыми для библиотеки и всех использующих её приложений.

- `ECPGget_PGconn(const char *`*`имя_подключения`*`)` возвращает указатель на подключение к базе данных, имеющее заданное имя. Если аргумент *`имя_подключения`* равен `NULL`, возвращается указатель на текущее подключение. Если определить подключение не удаётся, возвращается `NULL`. Полученный указатель на подключение, если требуется, можно использовать при вызове любых других функций libpq.

  > **Примечание**
  >
  > Манипулировать подключениями, открытыми средствами ecpg, напрямую через libpq не следует.

- `ECPGtransactionStatus(const char *имя_подключения)` возвращает состояние текущей транзакции для подключения, на которое указывает *имя_подключения*. Возвращаемые коды состояния перечислены в описании входящей в libpq функции `PQtransactionStatus()` в Разделе 31.2.
- `ECPGstatus(int номер_строки, const char* имя_подключения)` возвращает true при наличии подключения к базе данных и false в противном случае. В аргументе *имя_подключения* можно передать `NULL`, если применяется одно подключение.

33.12. Большие объекты

ECPG не поддерживает большие объекты напрямую, но приложение на базе ECPG может работать с большими объектами, используя предназначенные для этого функции, получив необходимый объект `PGconn` в результате вызова `ECPGget_PGconn()`. (Однако использовать функцию `ECPGget_PGconn()` и напрямую воздействовать на объекты `PGconn` следует очень осторожно; в идеале стоит исключить при этом другие обращения к базе данных через ECPG.)

Подробнее функция `ECPGget_PGconn()` описана в Разделе 33.11. Интерфейс функций для работы с большими объектами рассмотрен в Главе 32.

Функции для работы с большими объектами должны вызываться в блоке транзакций, поэтому если режим автофиксации отключён, необходимо явно выдавать команды `BEGIN`.

В Примере 33.2 приведён пример программы, показывающий, как создать, записать и прочитать большой объект в приложении ECPG.

Пример 33.2. Программа на базе ECPG, работающая с большими объектами

```
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>
#include <libpq/libpq-fs.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    PGconn     *conn;
    Oid         loid;
    int         fd;
    char        buf[256];
    int         buflen = 256;
    char        buf2[256];
    int         rc;

    memset(buf, 1, buflen);

    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    conn = ECPGget_PGconn("con1");
    printf("conn = %p\n", conn);

    /* create */
    loid = lo_create(conn, 0);
    if (loid &lt; 0)
        printf("lo_create() failed: %s", PQerrorMessage(conn));

    printf("loid = %d\n", loid);
```

```
    /* write test */
    fd = lo_open(conn, loid, INV_READ|INV_WRITE);
    if (fd &lt; 0)
        printf("lo_open() failed: %s", PQerrorMessage(conn));

    printf("fd = %d\n", fd);

    rc = lo_write(conn, fd, buf, buflen);
    if (rc &lt; 0)
        printf("lo_write() failed\n");

    rc = lo_close(conn, fd);
    if (rc &lt; 0)
        printf("lo_close() failed: %s", PQerrorMessage(conn));

    /* read test */
    fd = lo_open(conn, loid, INV_READ);
    if (fd &lt; 0)
        printf("lo_open() failed: %s", PQerrorMessage(conn));

    printf("fd = %d\n", fd);

    rc = lo_read(conn, fd, buf2, buflen);
    if (rc &lt; 0)
        printf("lo_read() failed\n");

    rc = lo_close(conn, fd);
    if (rc &lt; 0)
        printf("lo_close() failed: %s", PQerrorMessage(conn));

    /* check */
    rc = memcmp(buf, buf2, buflen);
    printf("memcmp() = %d\n", rc);

    /* cleanup */
    rc = lo_unlink(conn, loid);
    if (rc &lt; 0)
        printf("lo_unlink() failed: %s", PQerrorMessage(conn));

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

33.13. Приложения на C++

ECPG обеспечивает поддержку языка C++ в ограниченном объёме. Некоторые её особенности описаны в этом разделе.

Препроцессор `ecpg` принимает входной файл, написанный на C (или языке, подобном C) со встраиваемыми командами SQL, преобразует встроенные команды SQL в конструкции языка C и в результате формирует файл `.c`. Объявления библиотечных функций, вызываемых в конструкциях C, которые генерирует `ecpg`, заворачиваются в блоки `extern "C" { ... }` при использовании C++, так что они должны прозрачно работать в C++.

Однако вообще говоря, препроцессор `ecpg` понимает только C; он не воспринимает особый синтаксис и зарезервированные слова языка C++. Поэтому какой-то код SQL, встроенный в код

приложения на C++, в котором используются сложные особенности C++, может корректно не обработаться препроцессором или не работать как ожидается.

Надёжный подход к применению внедрённого кода SQL в приложении на C++ заключается в том, чтобы скрыть вызовы ECPG в модуле C, который будет вызываться приложением на C++ для работы с базой данных и который будет скомпонован с остальным кодом C++. Подробнее это описано в Подразделе 33.13.2.

33.13.1. Область видимости переменных среды

Препроцессор `ecpg` имеет понимание области видимости переменных в C. С языком C это довольно просто, так как область видимости переменных определяется их блоками кода. В C++, однако, переменные-члены класса задействуются не в том блоке кода, в каком они объявлены, так что препроцессор `ecpg` не сможет корректно определить область видимости таких переменных.

Например, в следующем случае препроцессор `ecpg` не сможет найти определение переменной `dbname` в методе `test`, так что произойдёт ошибка.

```
class TestCpp
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

  public:
    TestCpp();
    void test();
    ~TestCpp();
};

TestCpp::TestCpp()
{
    EXEC SQL CONNECT TO testdb1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
}

void Test::test()
{
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

TestCpp::~TestCpp()
{
    EXEC SQL DISCONNECT ALL;
}
```

При обработке данного кода будет выдано сообщение:

```
ecpg test_cpp.pgc
test_cpp.pgc:28: ERROR: variable "dbname" is not declared
```

(test_cpp.pgc:28: ОШИБКА: переменная "dbname" не объявлена)

Для решения этой проблемы можно немного изменить метод `test` и задействовать в нём локальную переменную для промежуточного хранения. Но предложенный подход нельзя считать хорошим, так как это портит код и снижает производительность.

```
void TestCpp::test()
{
    EXEC SQL BEGIN DECLARE SECTION;
```

```
    char tmp[1024];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT current_database() INTO :tmp;
    strlcpy(dbname, tmp, sizeof(tmp));

    printf("current_database = %s\n", dbname);
}
```

33.13.2. Разработка приложения на C++ с внешним модулем на C

Если вы поняли технические ограничения препроцессора `ecpg` с C++, вы можете прийти к заключению, что для использования ECPG в приложениях на C++ лучше связывать код C с кодом C++ на стадии компоновки, а не внедрять команды SQL непосредственно в код на C++. В данном разделе показывается, как отделить встраиваемые команды SQL от кода приложения на C++, на простом примере. В этом примере приложение реализуется на C++, а взаимодействие с сервером Postgres Pro построено на C и ECPG.

Для сборки нужно создать три типа файлов: файл на C (`*.pgc`), заголовочный файл и файл на C++:

`test_mod.pgc`

Модуль подпрограмм будет выполнять SQL-команды, встроенные в C. Этот код нужно будет преобразовать в `test_mod.c` с помощью препроцессора.

```
#include "test_mod.h"
#include <stdio.h>

void
db_connect()
{
    EXEC SQL CONNECT TO testdb1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;
}

void
db_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

void
db_disconnect()
{
    EXEC SQL DISCONNECT ALL;
}
```

`test_mod.h`

Заголовочный файл с объявлениями функций в модуле на языке C (`test_mod.pgc`). Он включается в `test_cpp.cpp`. Объявления в этом файле должны заключаться в блок `extern "C"`, так как он будет связываться с модулем C++.

```
#ifdef __cplusplus
extern "C" {
```

```
#endif

void db_connect();
void db_test();
void db_disconnect();

#ifdef __cplusplus
}
#endif
```

test_cpp.cpp

Основной код приложения, содержащий функцию main, а также, в данном примере, класс C++.

```
#include "test_mod.h"

class TestCpp
{
  public:
    TestCpp();
    void test();
    ~TestCpp();
};

TestCpp::TestCpp()
{
    db_connect();
}

void
TestCpp::test()
{
    db_test();
}

TestCpp::~TestCpp()
{
    db_disconnect();
}

int
main(void)
{
    TestCpp *t = new TestCpp();

    t->test();
    return 0;
}
```

Для сборки приложения проделайте следующее. Преобразуйте test_mod.pgc в test_mod.c с помощью ecpg, а затем получите test_mod.o, скомпилировав test_mod.c компилятором C:

```
ecpg -o test_mod.c test_mod.pgc
cc -c test_mod.c -o test_mod.o
```

После этого получите test_cpp.o, скомпилировав test_cpp.cpp компилятором C++:

```
c++ -c test_cpp.cpp -o test_cpp.o
```

Наконец, свяжите полученные объектные файлы, test_cpp.o и test_mod.o, в один исполняемый файл, выполнив компоновку под управлением компилятора C++:

```
c++ test_cpp.o test_mod.o -lecpg -o test_cpp
```

33.14. Команды встраиваемого SQL

В этом разделе описаны все команды, предназначенные специально для встраиваемого SQL. В Справке: «Команды SQL» также описаны обычные команды SQL, которые можно использовать и как встраиваемые, если явно не отмечено обратное.

ALLOCATE DESCRIPTOR

ALLOCATE DESCRIPTOR — выделить область SQL-дескриптора

Синтаксис

```
ALLOCATE DESCRIPTOR имя
```

Описание

`ALLOCATE DESCRIPTOR` выделяет новую именованную область SQL-дескриптора, через которую можно обмениваться данными между сервером Postgres Pro и программой на C.

После использования области дескрипторов должны освобождаться командой `DEALLOCATE DESCRIPTOR`.

Параметры

имя

Имя SQL-дескриптора, задаётся с учётом регистра. Это может быть идентификатор SQL или переменная среды C.

Примеры

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
```

Совместимость

Команда `ALLOCATE DESCRIPTOR` описана в стандарте SQL.

См. также

DEALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

CONNECT

CONNECT — установить подключение к базе данных

Синтаксис

```
CONNECT TO цель_подключения [ AS имя_подключения ] [ USER пользователь_подключения ]
CONNECT TO DEFAULT
CONNECT пользователь_подключения
DATABASE цель_подключения
```

Описание

Команда `CONNECT` устанавливает подключение клиента к серверу Postgres Pro.

Параметры

цель_подключения

цель_соединения задаёт целевой сервер и базу для подключения в одной из нескольких форм.

[*имя_бд*] [`@`*сервер*] [`:`*порт*]

Подключение по TCP/IP

`unix:postgresql://`*сервер* [`:`*порт*] `/` [*имя_бд*] [`?`*параметр_подключения*]

Подключение через Unix-сокеты

`tcp:postgresql://`*сервер* [`:`*порт*] `/` [*имя_бд*] [`?`*параметр_подключения*]

Подключение по TCP/IP

Строковая константа SQL

содержащая значение в одной из показанных выше форм

переменная среды C

переменная среды C типа `char[]` или `VARCHAR[]`, содержащая значение в одной из показанных выше форм

имя_подключения

Необязательный идентификатор подключения, позволяющий обращаться к этому подключению в других командах. Это может быть идентификатор SQL или переменная среды C.

пользователь_подключения

Имя пользователя для подключения к базе данных.

В этом параметре также можно передать имя и пароль одним из следующих способов: *имя_пользователя*`/`*пароль*, *имя_пользователя* `IDENTIFIED BY` *пароль* или *имя_пользователя* `USING` *пароль*.

В качестве имени пользователя и пароля можно задать идентификаторы SQL, строковые константы или переменные среды.

`DEFAULT`

Использовать все параметры подключения по умолчанию, которые определены библиотекой libpq.

Примеры

Несколько вариантов указания параметров подключения:

```
EXEC SQL CONNECT TO "connectdb" AS main;
EXEC SQL CONNECT TO "connectdb" AS second;
EXEC SQL CONNECT TO "unix:postgresql://200.46.204.71/connectdb" AS main USER
 connectuser;
EXEC SQL CONNECT TO "unix:postgresql://localhost/connectdb" AS main USER connectuser;
EXEC SQL CONNECT TO 'connectdb' AS main;
EXEC SQL CONNECT TO 'unix:postgresql://localhost/connectdb' AS main USER :user;
EXEC SQL CONNECT TO :db AS :id;
EXEC SQL CONNECT TO :db USER connectuser USING :pw;
EXEC SQL CONNECT TO @localhost AS main USER connectdb;
EXEC SQL CONNECT TO REGRESSDB1 as main;
EXEC SQL CONNECT TO AS main USER connectdb;
EXEC SQL CONNECT TO connectdb AS :id;
EXEC SQL CONNECT TO connectdb AS main USER connectuser/connectdb;
EXEC SQL CONNECT TO connectdb AS main;
EXEC SQL CONNECT TO connectdb@localhost AS main;
EXEC SQL CONNECT TO tcp:postgresql://localhost/ USER connectdb;
EXEC SQL CONNECT TO tcp:postgresql://localhost/connectdb USER connectuser IDENTIFIED BY
 connectpw;
EXEC SQL CONNECT TO tcp:postgresql://localhost:20/connectdb USER connectuser IDENTIFIED
 BY connectpw;
EXEC SQL CONNECT TO unix:postgresql://localhost/ AS main USER connectdb;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb AS main USER connectuser;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser IDENTIFIED
 BY "connectpw";
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser USING
 "connectpw";
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb?connect_timeout=14 USER
 connectuser;
```

Следующий пример программы демонстрирует применение переменных среды для определения параметров подключения:

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    char *dbname     = "testdb";    /* имя базы данных */
    char *user       = "testuser";  /* имя пользователя подключения */
    char *connection = "tcp:postgresql://localhost:5432/testdb";
                                    /* строка подключения */
    char ver[256];                  /* буфер для хранения строки версии */
EXEC SQL END DECLARE SECTION;

    ECPGdebug(1, stderr);

    EXEC SQL CONNECT TO :dbname USER :user;
    EXEC SQL SELECT pgpro_version() INTO :ver;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL DISCONNECT;

    printf("version: %s\n", ver);

    EXEC SQL CONNECT TO :connection USER :user;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL SELECT pgpro_version() INTO :ver;
    EXEC SQL DISCONNECT;

    printf("version: %s\n", ver);
```

```
    return 0;
}
```

Совместимость

Команда `CONNECT` описана в стандарте SQL, но формат параметров подключения определяется реализацией.

См. также

DISCONNECT, SET CONNECTION

DEALLOCATE DESCRIPTOR

DEALLOCATE DESCRIPTOR — освободить область SQL-дескриптора

Синтаксис

```
DEALLOCATE DESCRIPTOR имя
```

Описание

`DEALLOCATE DESCRIPTOR` освобождает область именованного SQL-дескриптора.

Параметры

имя

Имя дескриптора, подлежащего освобождению, задаётся с учётом регистра. Это может быть идентификатор SQL или переменная среды C.

Примеры

```
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

Совместимость

Команда `DEALLOCATE DESCRIPTOR` описана в стандарте SQL.

См. также

ALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

DECLARE

DECLARE — определить курсор

Синтаксис

```
DECLARE имя_курсора [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH |
 WITHOUT } HOLD ] FOR подготовленный_оператор
DECLARE имя_курсора [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH |
 WITHOUT } HOLD ] FOR запрос
```

Описание

`DECLARE` объявляет курсор для прохода по набору результатов подготовленного оператора. Эта команда несколько отличается от обычной SQL-команды `DECLARE`: тогда как последняя выполняет запрос и подготавливает набор результатов для получения, встраиваемая SQL-команда просто объявляет имя в качестве «переменной цикла» для прохода по набору результатов запроса; фактически запрос выполнится, когда курсор будет открыт командой `OPEN`.

Параметры

имя_курсора

Имя курсора, задаётся с учётом регистра. Это может быть идентификатор SQL или переменная среды C.

подготовленный_оператор

Имя подготовленного запроса, задаваемое SQL-идентификатором или переменной среды.

запрос

Команда SELECT или VALUES, выдающая строки, которые будут получены через курсор.

Параметры курсора рассматриваются в описании DECLARE.

Примеры

Примеры объявления курсора для запроса:

```
EXEC SQL DECLARE C CURSOR FOR SELECT * FROM My_Table;
EXEC SQL DECLARE C CURSOR FOR SELECT Item1 FROM T;
EXEC SQL DECLARE cur1 CURSOR FOR SELECT pgpro_version();
```

Пример объявления курсора для подготовленного оператора:

```
EXEC SQL PREPARE stmt1 AS SELECT pgpro_version();
EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

Совместимость

Команда `DECLARE` описана в стандарте SQL.

См. также

OPEN, CLOSE, DECLARE

DESCRIBE

DESCRIBE — получить информацию о подготовленном операторе или наборе результатов

Синтаксис

```
DESCRIBE [ OUTPUT ] подготовленный_оператор USING [ SQL ] DESCRIPTOR имя_дескриптора
DESCRIBE [ OUTPUT ] подготовленный_оператор INTO [ SQL ] DESCRIPTOR имя_дескриптора
DESCRIBE [ OUTPUT ] подготовленный_оператор INTO имя_sqlda
```

Описание

`DESCRIBE` получает метаданные о результирующих столбцах, содержащихся в подготовленном операторе, не считывая собственно строки результата.

Параметры

подготовленный_оператор

Имя подготовленного оператора. Это может быть идентификатор SQL или переменная среды C.

имя_дескриптора

Имя дескриптора, задаётся с учётом регистра. Это может быть идентификатор SQL или переменная среды C.

имя_sqlda

Имя переменной SQLDA.

Примеры

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :charvar = NAME;
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

Совместимость

Команда `DESCRIBE` описана в стандарте SQL.

См. также

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

DISCONNECT

DISCONNECT — закрыть подключение к базе данных

Синтаксис

```
DISCONNECT имя_подключения
DISCONNECT [ CURRENT ]
DISCONNECT DEFAULT
DISCONNECT ALL
```

Описание

`DISCONNECT` закрывает подключение (или все подключения) к базе данных.

Параметры

имя_подключения

Имя подключения к базе данных устанавливается командой `CONNECT`.

`CURRENT`

Закрывает «текущее» подключение, то есть подключение, открытое последним, либо установленное командой `SET CONNECTION`. Текущее подключение подразумевается по умолчанию, если `DISCONNECT` выполняется без аргументов.

`DEFAULT`

Закрывает подключение по умолчанию.

`ALL`

Закрывает все открытые подключения.

Примеры

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS DEFAULT USER testuser;
    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL CONNECT TO testdb AS con2 USER testuser;
    EXEC SQL CONNECT TO testdb AS con3 USER testuser;

    EXEC SQL DISCONNECT CURRENT;  /* закрывает con3          */
    EXEC SQL DISCONNECT DEFAULT;  /* закрывает DEFAULT       */
    EXEC SQL DISCONNECT ALL;      /* закрывает con2 и con1 */

    return 0;
}
```

Совместимость

Команда `DISCONNECT` описана в стандарте SQL.

См. также

CONNECT, SET CONNECTION

EXECUTE IMMEDIATE

EXECUTE IMMEDIATE — динамически подготовить и выполнить оператор

Синтаксис

```
EXECUTE IMMEDIATE строка
```

Описание

`EXECUTE IMMEDIATE` немедленно подготавливает и выполняет динамически задаваемый SQL-оператор, не получая при этом строки результата.

Параметры

строка

Строковая константа C или переменная среды C, содержащая SQL-оператор, который нужно выполнить.

Примеры

Пример выполнения оператора `INSERT` с применением команды `EXECUTE IMMEDIATE` и переменной среды C `command`:

```
sprintf(command, "INSERT INTO test (name, amount, letter) VALUES ('db: ''r1''', 1,
 'f')");
EXEC SQL EXECUTE IMMEDIATE :command;
```

Совместимость

Команда `EXECUTE IMMEDIATE` описана в стандарте SQL.

GET DESCRIPTOR

GET DESCRIPTOR — получить информацию из области дескриптора SQL

Синтаксис

```
GET DESCRIPTOR имя_дескриптора :cvariable = элемент_заголовка_дескриптора [, ... ]
GET DESCRIPTOR имя_дескриптора VALUE номер_столбца :cvariable = элемент_дескриптора
 [, ... ]
```

Описание

`GET DESCRIPTOR` получает информацию о наборе результатов запроса из области дескриптора SQL и сохраняет её в переменные среды. Область дескриптора обычно предварительно заполняется командами `FETCH` или `SELECT`, чтобы из неё можно было перенести сопутствующую информацию в переменные среды.

Эта команда имеет две формы: первая форма выдаёт элементы из «заголовка» дескриптора, который относится ко всему набору результатов в целом. Например, это число строк. Другая форма, требующая указания в дополнительном параметре номера столбца, выдаёт информацию о конкретном столбце строки. В качестве примеров можно привести имя столбца и фактическое значение в этом столбце.

Параметры

имя_дескриптора

Имя дескриптора.

элемент_заголовка_дескриптора

Идентификатор, определяющий, какой элемент заголовка нужно получить. В настоящее время поддерживается только `COUNT`, позволяющий получить число столбцов в наборе результатов.

номер_столбца

Номер столбца, информацию о котором нужно получить. Нумерация начинается с 1.

элемент_дескриптора

Идентификатор, определяющий, какой элемент информации о столбце нужно получить. Список поддерживаемых элементов приведён в Подразделе 33.7.1.

cvariable

Переменная среды C, в которую будут сохранены данные, полученные из области дескриптора.

Примеры

Пример получения числа столбцов в наборе результатов:

```
EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
```

Пример получения размера данных в первом столбце:

```
EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length = RETURNED_OCTET_LENGTH;
```

Пример получения содержимого данных второго столбца в виде строки:

```
EXEC SQL GET DESCRIPTOR d VALUE 2 :d_data = DATA;
```

Следующий пример демонстрирует всю процедуру выполнения `SELECT current_database();` и вывода числа столбцов, длины данных в столбце и содержимого столбца:

```
int
```

```
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int  d_count;
    char d_data[1024];
    int  d_returned_octet_length;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL ALLOCATE DESCRIPTOR d;

    /* Объявить, открыть курсор и присвоить ему дескриптор */
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database();
    EXEC SQL OPEN cur;
    EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;

    /* Получить общее число столбцов */
    EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
    printf("d_count                 = %d\n", d_count);

    /* Получить размер возвращённого столбца */
    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length = RETURNED_OCTET_LENGTH;
    printf("d_returned_octet_length = %d\n", d_returned_octet_length);

    /* Выбрать возвращённый столбец в виде текстовой строки */
    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_data = DATA;
    printf("d_data                  = %s\n", d_data);

    /* Закрытие */
    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;

    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

При выполнении этого примера будет получен такой результат:

```
d_count                 = 1
d_returned_octet_length = 6
d_data                  = testdb
```

Совместимость

Команда `GET DESCRIPTOR` описана в стандарте SQL.

См. также

ALLOCATE DESCRIPTOR, SET DESCRIPTOR

OPEN

OPEN — открыть динамический курсор

Синтаксис

```
OPEN имя_курсора
OPEN имя_курсора USING значение [, ... ]
OPEN имя_курсора USING SQL DESCRIPTOR имя_дескриптора
```

Описание

`OPEN` открывает курсор и в дополнение может связывать фактические значения с местозаполнителями в объявлении курсора. Курсор должен быть предварительно объявлен командой `DECLARE`. Команда `OPEN` запускает выполнение запроса на сервере.

Параметры

имя_курсора

Имя открываемого курсора. Этот может быть идентификатор SQL или переменная среды.

значение

Значение, связываемое с местозаполнителем в курсоре. Это может быть константа SQL, переменная среды C или переменная среды с индикатором.

имя_дескриптора

Имя дескриптора, содержащего значения, которые должны быть связаны с местозаполнителями в курсоре. Это может быть идентификатор SQL или переменная среды C.

Примеры

```
EXEC SQL OPEN a;
EXEC SQL OPEN d USING 1, 'test';
EXEC SQL OPEN c1 USING SQL DESCRIPTOR mydesc;
EXEC SQL OPEN :curname1;
```

Совместимость

Команда `OPEN` описана в стандарте SQL.

См. также

DECLARE, CLOSE

PREPARE

PREPARE — подготовить оператор к выполнению

Синтаксис

```
PREPARE имя FROM строка
```

Описание

Команда `PREPARE` подготавливает к выполнению динамический оператор, задаваемый в виде строки. Она отличается от обычного SQL-оператора PREPARE, который также можно использовать во встраиваемых командах. Для обоих типов подготовленных операторов применяется команда EXECUTE.

Параметры

подготовленный_оператор

Идентификатор для подготовленного запроса.

строка

Строковая константа C или переменная среды C, содержащая подготавливаемый оператор: SELECT, INSERT, UPDATE или DELETE.

Примеры

```
char *stmt = "SELECT * FROM test1 WHERE a = ? AND b = ?";

EXEC SQL ALLOCATE DESCRIPTOR outdesc;
EXEC SQL PREPARE foo FROM :stmt;

EXEC SQL EXECUTE foo USING SQL DESCRIPTOR indesc INTO SQL DESCRIPTOR outdesc;
```

Совместимость

Команда `PREPARE` описана в стандарте SQL.

См. также

EXECUTE

SET AUTOCOMMIT

SET AUTOCOMMIT — установить режим автофиксации для текущего сеанса

Синтаксис

```
SET AUTOCOMMIT { = | TO } { ON | OFF }
```

Описание

`SET AUTOCOMMIT` устанавливает режим автофиксации для текущего сеанса использования базы данных. По умолчанию программы со встраиваемым SQL работают *не* в режиме автофиксации, так что в определённые моменты нужно явно выполнять `COMMIT`. Эта команда может переключить сеанс в режим автофиксации, когда неявно фиксируется каждый отдельный оператор.

Совместимость

`SET AUTOCOMMIT` — расширение Postgres Pro ECPG.

SET CONNECTION

SET CONNECTION — выбрать подключение к базе данных

Синтаксис

```
SET CONNECTION [ TO | = ] имя_подключения
```

Описание

`SET CONNECTION` устанавливает «текущее» подключение к базе данных, которое будет использоваться командами, не задающими подключение явно.

Параметры

имя_подключения

Имя подключения к базе данных устанавливается командой `CONNECT`.

`DEFAULT`

Устанавливает заданное подключение подключением по умолчанию.

Примеры

```
EXEC SQL SET CONNECTION TO con2;
EXEC SQL SET CONNECTION = con1;
```

Совместимость

Команда `SET CONNECTION` описана в стандарте SQL.

См. также

CONNECT, DISCONNECT

SET DESCRIPTOR

SET DESCRIPTOR — внести информацию в область дескриптора SQL

Синтаксис

```
SET DESCRIPTOR имя_дескриптора элемент_заголовка_дескриптора = значение [, ... ]
SET DESCRIPTOR имя_дескриптора VALUE номер элемент_дескриптора = значение [, ...]
```

Описание

`SET DESCRIPTOR` заполняет область SQL-дескриптора значениями. Заполненная область дескриптора обычно применяется для привязывания параметров при выполнении подготовленного запроса.

Эта команда имеет две формы: первая применяется к «заголовку» дескриптора, который не зависит от конкретных данных. Вторая форма устанавливает значения для определённых полей по номерам.

Параметры

имя_дескриптора

Имя дескриптора.

элемент_заголовка_дескриптора

Идентификатор, определяющий, какой элемент заголовка нужно задать. В настоящее время поддерживается только `COUNT`, позволяющий задать число элементов в дескрипторе.

номер

Номер элемента дескриптора, для которого задаётся значение. Нумерация начинается с 1.

элемент_дескриптора

Идентификатор, определяющий, какой элемент нужно установить в дескрипторе. Список поддерживаемых элементов приведён в Подразделе 33.7.1.

значение

Значение, которое нужно поместить в элемент дескриптора. Это может быть константа SQL или переменная среды C.

Примеры

```
EXEC SQL SET DESCRIPTOR indesc COUNT = 1;
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = 2;
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = :val1;
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val1, DATA = 'some string';
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val2null, DATA = :val2;
```

Совместимость

Команда `SET DESCRIPTOR` описана в стандарте SQL.

См. также

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

TYPE

TYPE — создать новый тип данных

Синтаксис

```
TYPE имя_типа IS тип_C
```

Описание

Команда `TYPE` определяет новый тип C. Она равнозначна добавлению `typedef` в секции объявлений.

Эта команда принимается, только когда `ecpg` запускается с параметром `-c`.

Параметры

имя_типа

Имя нового типа. Это имя должно быть допустимым для типа в языке C.

тип_C

Определение типа C.

Примеры

```
EXEC SQL TYPE customer IS
    struct
    {
        varchar name[50];
        int     phone;
    };

EXEC SQL TYPE cust_ind IS
    struct ind
    {
        short   name_ind;
        short   phone_ind;
    };

EXEC SQL TYPE c IS char reference;
EXEC SQL TYPE ind IS union { int integer; short smallint; };
EXEC SQL TYPE intarray IS int[AMOUNT];
EXEC SQL TYPE str IS varchar[BUFFERSIZ];
EXEC SQL TYPE string IS char[11];
```

Пример программы, в которой используется `EXEC SQL TYPE`:

```
EXEC SQL WHENEVER SQLERROR SQLPRINT;

EXEC SQL TYPE tt IS
    struct
    {
        varchar v[256];
        int     i;
    };

EXEC SQL TYPE tt_ind IS
    struct ind {
        short   v_ind;
        short   i_ind;
    };
```

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    tt t;
    tt_ind t_ind;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    EXEC SQL SELECT current_database(), 256 INTO :t:t_ind LIMIT 1;

    printf("t.v = %s\n", t.v.arr);
    printf("t.i = %d\n", t.i);

    printf("t_ind.v_ind = %d\n", t_ind.v_ind);
    printf("t_ind.i_ind = %d\n", t_ind.i_ind);

    EXEC SQL DISCONNECT con1;

    return 0;
}
```

Эта программа выдаёт следующее:

```
t.v = testdb
t.i = 256
t_ind.v_ind = 0
t_ind.i_ind = 0
```

Совместимость

Команда `TYPE` — расширение Postgres Pro.

VAR

VAR — определить переменную

Синтаксис

```
VAR имя_переменной IS тип_C
```

Описание

Команда `VAR` назначает переменной среды C новый тип данных C. Эта переменная среды должна быть объявлена ранее в секции объявлений.

Параметры

имя_переменной

Имя переменной C.

тип_C

Определение типа C.

Примеры

```
Exec sql begin declare section;
short a;
exec sql end declare section;
EXEC SQL VAR a IS int;
```

Совместимость

Команда `VAR` — расширение Postgres Pro.

WHENEVER

WHENEVER — определить действие, которое должно выполняться, когда при обработке SQL-оператора возникает определённое условие

Синтаксис

```
WHENEVER { NOT FOUND | SQLERROR | SQLWARNING } действие
```

Описание

Устанавливает поведение в случае определённых условий (строки не найдены, выданы предупреждения или ошибки SQL и т. д.), возникающих в ходе выполнения SQL.

Параметры

Описание параметров приведено в Подразделе 33.8.1.

Примеры

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLWARNING DO warn();
EXEC SQL WHENEVER SQLERROR sqlprint;
EXEC SQL WHENEVER SQLERROR CALL print2();
EXEC SQL WHENEVER SQLERROR DO handle_error("select");
EXEC SQL WHENEVER SQLERROR DO sqlnotice(NULL, NONO);
EXEC SQL WHENEVER SQLERROR DO sqlprint();
EXEC SQL WHENEVER SQLERROR GOTO error_label;
EXEC SQL WHENEVER SQLERROR STOP;
```

Типичное применение конструкция `WHENEVER NOT FOUND BREAK` находит в обработке результатов запроса в цикле:

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL ALLOCATE DESCRIPTOR d;
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database(), 'hoge', 256;
    EXEC SQL OPEN cur;

    /* по достижении конца набора результатов прервать цикл while */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;
        ...
    }

    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;

    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;

    return 0;
```

```
}
```

Совместимость

Команда `WHENEVER` описана в стандарте SQL, но большинство действий относятся к расширениям Postgres Pro.

33.15. Режим совместимости с Informix

Препроцессор `ecpg` может работать в так называемом *режиме совместимости с Informix*. Если этот режим включён, `ecpg` старается работать как предкомпилятор Informix для кода Informix E/SQL. Вообще говоря, это позволяет записывать встраиваемые команды SQL, используя знак доллара вместо слов `EXEC SQL`:

```
$int j = 3;
$CONNECT TO :dbname;
$CREATE TABLE test(i INT PRIMARY KEY, j INT);
$INSERT INTO test(i, j) VALUES (7, :j);
$COMMIT;
```

> **Примечание**
>
> Между `$` и последующей директивой препроцессора (в частности, `include`, `define`, `ifdef` и т. п.) не должно быть пробельных символов. В противном случае препроцессор воспримет следующее слово как имя переменной среды.

Поддерживаются два режима совместимости: `INFORMIX` и `INFORMIX_SE`

При компоновке программ, использующих этот режим совместимости, обязательно подключите библиотеку `libcompat`, поставляемую с ECPG.

Помимо ранее упомянутого синтаксического сахара, режим совместимости с Informix приносит из E/SQL в ECPG набор функций для ввода, вывода и преобразования данных, а также встраиваемые операторы SQL.

Режим совместимости с Informix тесно связан с библиотекой pgtypeslib из ECPG. Библиотека pgtypeslib сопоставляет типы данных SQL с типами данных в ведущей программе на C, а большинство дополнительных функций режима совместимости с Informix позволяют работать с этими типами C. Заметьте, однако, что степень совместимости ограничена. ECPG не пытается копировать поведение Informix; вы можете выполнять примерно те же операции и пользоваться функции с теми же именами и с тем же поведением, но если вы используете Informix, просто заменить одно средство другим на данный момент нельзя. Более того, есть различия и в типах данных. В частности, типы даты и интервала в Postgres Pro не воспринимают диапазоны, как например, `YEAR TO MINUTE`, так что и в ECPG это не будет поддерживаться.

33.15.1. Дополнительные типы

Теперь в режиме Informix без указания `typedef` поддерживается специальный псевдотип Informix "string" для хранения символьной строки, обрезаемой справа. На самом деле, в режиме Informix ECPG откажется обрабатывать исходные файлы, содержащие определение типа `typedef некоторый_тип string;`

```
EXEC SQL BEGIN DECLARE SECTION;
string userid; /* эта переменная будет содержать обрезанные данные */
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH MYCUR INTO :userid;
```

33.15.2. Дополнительные/недостающие операторы встраиваемого SQL

`CLOSE DATABASE`

Этот оператор закрывает текущее подключение. Фактически это синоним команды `DISCONNECT CURRENT` в ECPG:

```
$CLOSE DATABASE;                /* закрыть текущее подключение */
EXEC SQL CLOSE DATABASE;
```

`FREE имя_курсора`

Из-за различий в подходах ECPG и ESQL/C Informix (т. е. другого разделения на чисто грамматические преобразования и вызовы нижележащей библиотеки времени выполнения), в ECPG нет оператора `FREE имя_курсора`. Это связано с тем, что в ECPG команда `DECLARE CURSOR` не сводится к вызову функции в библиотеке времени выполнения, которая бы принимала имя курсора. Это значит, что курсоры SQL в библиотеке ECPG не требуют обслуживания, оно требуется только на уровне сервера Postgres Pro.

`FREE имя_оператора`

Команда `FREE имя_оператора` является синонимом команды `DEALLOCATE PREPARE имя_оператора`.

33.15.3. Области дескрипторов SQLDA, совместимые с Informix

Режим совместимости с Informix поддерживает структуру, отличную от описанной в Подразделе 33.7.2. См. ниже:

```
struct sqlvar_compat
{
    short   sqltype;
    int     sqllen;
    char   *sqldata;
    short  *sqlind;
    char   *sqlname;
    char   *sqlformat;
    short   sqlitype;
    short   sqlilen;
    char   *sqlidata;
    int     sqlxid;
    char   *sqltypename;
    short   sqltypelen;
    short   sqlownerlen;
    short   sqlsourcetype;
    char   *sqlowername;
    int     sqlsourceid;
    char   *sqlilongdata;
    int     sqlflags;
    void   *sqlreserved;
};

struct sqlda_compat
{
    short  sqld;
    struct sqlvar_compat *sqlvar;
    char   desc_name[19];
    short  desc_occ;
    struct sqlda_compat *desc_next;
    void  *reserved;
```

```
};

typedef struct sqlvar_compat    sqlvar_t;
typedef struct sqlda_compat     sqlda_t;
```

Глобальные свойства:

`sqld`

Число полей в дескрипторе `SQLDA`.

`sqlvar`

Указатель на свойства по полям.

`desc_name`

Не используется, заполняется нулями.

`desc_occ`

Размер структуры в памяти.

`desc_next`

Указатель на следующую структуру SQLDA, если набор результатов содержит больше одной записи.

`reserved`

Неиспользуемый указатель, содержит NULL. Сохраняется для совместимости с Informix.

Свойства, относящиеся к полям, описаны ниже, они хранятся в массиве `sqlvar`:

`sqltype`

Тип поля. Соответствующие константы представлены в `sqltypes.h`

`sqllen`

Длина данных поля.

`sqldata`

Указатель на данные поля. Этот указатель имеет тип `char *`, но он указывает на данные в двоичном формате. Например:

```
int intval;

switch (sqldata->sqlvar[i].sqltype)
{
    case SQLINTEGER:
        intval = *(int *)sqldata->sqlvar[i].sqldata;
        break;
  ...
}
```

`sqlind`

Указатель на индикатор NULL. Если возвращается командами DESCRIBE или FETCH, это всегда действительный указатель. Если передаётся на вход команде `EXECUTE ... USING sqlda;`, NULL вместо указателя означает, что значение этого поля отлично от NULL. Чтобы обозначить NULL в поле, необходимо корректно установить этот указатель и `sqlitype`. Например:

```
if (*(int2 *)sqldata->sqlvar[i].sqlind != 0)
    printf("value is NULL\n");
```

`sqlname`

Имя поля, в виде строки с завершающим 0.

`sqlformat`

Зарезервировано в Informix, значение `PQfformat()` для данного поля.

`sqlitype`

Тип данных индикатора NULL. При получении данных с сервера это всегда SQLSMINT. Когда `SQLDA` используется в параметризованном запросе, данные индикатора обрабатываются в соответствии с указанным здесь типом.

`sqlilen`

Длина данных индикатора NULL.

`sqlxid`

Расширенный тип поля, результат функции `PQftype()`.

`sqltypename`
`sqltypelen`
`sqlownerlen`
`sqlsourcetype`
`sqlownername`
`sqlsourceid`
`sqlflags`
`sqlreserved`

Не используются.

`sqlilongdata`

Совпадает с `sqldata`, если `sqllen` превышает 32 Кбайта.

Например:

```
EXEC SQL INCLUDE sqlda.h;

    sqlda_t        *sqlda; /* Это объявление не обязательно должно быть внутри DECLARE
 SECTION */

    EXEC SQL BEGIN DECLARE SECTION;
    char *prep_stmt = "select * from table1";
    int i;
    EXEC SQL END DECLARE SECTION;

    ...

    EXEC SQL PREPARE mystmt FROM :prep_stmt;

    EXEC SQL DESCRIBE mystmt INTO sqlda;

    printf("# of fields: %d\n", sqlda->sqld);
    for (i = 0; i < sqlda->sqld; i++)
      printf("field %d: \"%s\"\n", sqlda->sqlvar[i]->sqlname);

    EXEC SQL DECLARE mycursor CURSOR FOR mystmt;
    EXEC SQL OPEN mycursor;
    EXEC SQL WHENEVER NOT FOUND GOTO out;
```

```
while (1)
{
  EXEC SQL FETCH mycursor USING sqlda;
}

EXEC SQL CLOSE mycursor;

free(sqlda); /* Освобождать нужно только основную структуру,
              * sqlda и sqlda->sqlvar находятся в одной выделенной области. */
```

Дополнительную информацию можно найти в заголовочном файле `sqlda.h` и в регрессионном тесте `src/interfaces/ecpg/test/compat_informix/sqlda.pgc`.

33.15.4. Дополнительные функции

`decadd`

Складывает два значения типа decimal.

```
int decadd(decimal *arg1, decimal *arg2, decimal *sum);
```

Эта функция получает указатель на первый операнд типа decimal (`arg1`), указатель на второй операнд типа decimal (`arg2`) и указатель на переменную типа decimal, в которую будет записана сумма (`sum`). В случае успеха эта функция возвращает 0. `ECPG_INFORMIX_NUM_OVERFLOW` возвращается в случае переполнения, а `ECPG_INFORMIX_NUM_UNDERFLOW` в случае антипереполнения. При любых других ошибках возвращается -1, а в `errno` устанавливается код `errno` из pgtypeslib.

`deccmp`

Сравнивает два значения типа decimal.

```
int deccmp(decimal *arg1, decimal *arg2);
```

Эта функция получает указатель на первое значение типа decimal (`arg1`), указатель на второе значение типа decimal (`arg2`) и возвращает целое, отражающее результат сравнения этих чисел.

- 1, если значение, на которое указывает `arg1`, больше значения, на которое указывает `var2`
- -1, если значение, на которое указывает `arg1`, меньше значения, на которое указывает `arg2`
- 0, если значение, на которое указывает `arg1`, равно значению, на которое указывает `arg2`

`deccopy`

Копирует значение типа decimal.

```
void deccopy(decimal *src, decimal *target);
```

Функция принимает в первом аргументе (`src`) указатель на значение decimal, которое должно быть скопировано, а во втором аргументе (`target`) принимает указатель на структуру типа decimal для скопированного значения.

`deccvasc`

Преобразует значение из представления ASCII в тип decimal.

```
int deccvasc(char *cp, int len, decimal *np);
```

Эта функция получает указатель на строку, содержащую строковое представление числа, которое нужно преобразовать, (`cp`), а также его длину `len`. В `np` передаётся указатель на переменную типа decimal, в которую будет помещён результат преобразования.

Допустимыми являются, например следующие форматы: `-2`, `.794`, `+3.44`, `592.49E07` или `-32.84e-4`.

В случае успеха эта функция возвращает 0. При переполнении или антипереполнении возвращается ECPG_INFORMIX_NUM_OVERFLOW или ECPG_INFORMIX_NUM_UNDERFLOW, соответственно. Если разобрать ASCII-представление не удаётся, возвращается ECPG_INFORMIX_BAD_NUMERIC или ECPG_INFORMIX_BAD_EXPONENT, если не удаётся разобрать компонент экспоненты.

deccvdbl

Преобразует значение double в значение типа decimal.

```
int deccvdbl(double dbl, decimal *np);
```

Данная функция принимает в первом аргументе (dbl) переменную типа double, которая должна быть преобразована. Во втором аргументе (np) она принимает указатель на переменную decimal, в которую будет помещён результат операции.

Эта функция возвращает 0 в случае успеха, либо отрицательное значение, если выполнить преобразование не удалось.

deccvint

Преобразует значение int в значение типа decimal.

```
int deccvint(int in, decimal *np);
```

Данная функция принимает в первом аргументе (in) переменную типа int, которая должна быть преобразована. Во втором аргументе (np) она принимает указатель на переменную decimal, в которую будет помещён результат операции.

Эта функция возвращает 0 в случае успеха, либо отрицательное значение, если выполнить преобразование не удалось.

deccvlong

Преобразует значение long в значение типа decimal.

```
int deccvlong(long lng, decimal *np);
```

Данная функция принимает в первом аргументе (lng) переменную типа long, которая должна быть преобразована. Во втором аргументе (np) она принимает указатель на переменную decimal, в которую будет помещён результат операции.

Эта функция возвращает 0 в случае успеха, либо отрицательное значение, если выполнить преобразование не удалось.

decdiv

Делит одну переменную типа decimal на другую.

```
int decdiv(decimal *n1, decimal *n2, decimal *result);
```

Эта функция получает указатели на переменные (n1 и n2) и вычисляет частное n1/n2. В result передаётся указатель на переменную, в которую будет помещён результат операции.

В случае успеха возвращается 0, а при ошибке — отрицательное значение. В случае переполнения или антипереполнения данная функция возвращает ECPG_INFORMIX_NUM_OVERFLOW или ECPG_INFORMIX_NUM_UNDERFLOW, соответственно. При попытке деления на ноль возвращается ECPG_INFORMIX_DIVIDE_ZERO.

decmul

Перемножает два значения типа decimal.

```
int decmul(decimal *n1, decimal *n2, decimal *result);
```

Эта функция получает указатели на переменные (n1 и n2) и вычисляет произведение n1*n2. В result передаётся указатель на переменную, в которую будет помещён результат операции.

В случае успеха возвращается 0, а при ошибке — отрицательное значение. В случае переполнения или антипереполнения данная функция возвращает ECPG_INFORMIX_NUM_OVERFLOW или ECPG_INFORMIX_NUM_UNDERFLOW, соответственно.

decsub

Вычитает одно значение типа decimal из другого.

```
int decsub(decimal *n1, decimal *n2, decimal *result);
```

Эта функция получает указатели на переменные (n1 и n2) и вычисляет разность n1-n2. В result передаётся указатель на переменную, в которую будет помещён результат операции.

В случае успеха возвращается 0, а при ошибке — отрицательное значение. В случае переполнения или антипереполнения данная функция возвращает ECPG_INFORMIX_NUM_OVERFLOW или ECPG_INFORMIX_NUM_UNDERFLOW, соответственно.

dectoasc

Преобразует переменную типа decimal в представление ASCII (в строку C char*).

```
int dectoasc(decimal *np, char *cp, int len, int right)
```

Эта функция получает указатель на переменную типа decimal (np), которая будет преобразована в текстовое представление. Аргумент cp указывает на буфер, в который будет помещён результат операции. Аргумент right определяет, сколько должно выводиться цифр правее десятичной точки. Результат будет округлён до этого числа десятичных цифр. Значение right, равное -1, указывает, что выводиться должны все имеющиеся десятичные цифры. Если длина выходного буфера, которую задаёт len, недостаточна для помещения в него текстового представления, включая завершающий нулевой байт, в буфере сохраняется один знак * и возвращается -1.

Эта функция возвращает -1, если буфер cp слишком мал, либо ECPG_INFORMIX_OUT_OF_MEMORY при нехватке памяти.

dectodbl

Преобразует переменную типа decimal в тип double.

```
int dectodbl(decimal *np, double *dblp);
```

Эта функция получает указатель (np) на значение decimal, которое нужно преобразовать, и указатель (dblp) на переменную double, в которую будет помещён результат операции.

В случае успеха возвращается 0, или отрицательное значение, если выполнить преобразование не удалось.

dectoint

Преобразует переменную типа decimal в тип integer.

```
int dectoint(decimal *np, int *ip);
```

Эта функция получает указатель (np) на значение decimal, которое нужно преобразовать, и указатель (ip) на целочисленную переменную, в которую будет помещён результат операции.

В случае успеха возвращается 0, или отрицательное значение, если выполнить преобразование не удалось. В случае переполнения возвращается ECPG_INFORMIX_NUM_OVERFLOW.

Заметьте, что реализация ECPG отличается от реализации Informix. В Informix целое ограничивается диапазоном -32767 .. 32767, тогда как в ECPG ограничение зависит от архитектуры (INT_MIN .. INT_MAX).

dectolong

Преобразует переменную типа decimal в тип long.

```
int dectolong(decimal *np, long *lngp);
```

Эта функция получает указатель (`np`) на значение decimal, которое нужно преобразовать, и указатель (`lngp`) на переменную типа long, в которую будет помещён результат операции.

В случае успеха возвращается 0, или отрицательное значение, если выполнить преобразование не удалось. В случае переполнения возвращается `ECPG_INFORMIX_NUM_OVERFLOW`.

Заметьте, что реализация ECPG отличается от реализации Informix. В Informix длинное целое ограничено диапазоном -2 147 483 647 .. 2 147 483 647, тогда как в ECPG ограничение зависит от архитектуры (`-LONG_MAX .. LONG_MAX`).

`rdatestr`

Преобразует дату в строку C char*.

```
int rdatestr(date d, char *str);
```

Эта функция принимает два аргумента. В первом (`d`) передаётся дата, которую нужно преобразовать, а во втором указатель на целевую строку. Результат всегда выводится в формате `yyyy-mm-dd`, так что для этой строки нужно выделить минимум 11 байт (включая завершающий нулевой байт).

Эта функция возвращает 0 в случае успеха, а в случае ошибки — отрицательное значение.

Заметьте, что реализация ECPG отличается от реализации Informix. В Informix формат вывода можно изменить переменными окружения, а в ECPG он фиксирован.

`rstrdate`

Разбирает текстовое представление даты.

```
int rstrdate(char *str, date *d);
```

Эта функция получает текстовое представление (`str`) даты, которую нужно преобразовать, и указатель на переменную типа date (`d`). Для данной функции нельзя задать маску формата. Она использует стандартную маску формата Informix, а именно: `mm/dd/yyyy`. Внутри эта функция вызывает `rdefmtdate`. Таким образом, `rstrdate` не будет быстрее, и если у вас есть выбор, используйте функцию `rdefmtdate`, которая позволяет явно задать маску формата.

Эта функция возвращает те же значения, что и `rdefmtdate`.

`rtoday`

Выдаёт текущую дату.

```
void rtoday(date *d);
```

Эта функция получает указатель на переменную (`d`) типа date, в которую будет записана текущая дата.

Внутри эта функция вызывает `PGTYPESdate_today`.

`rjulmdy`

Извлекает значения дня, месяца и года из переменной типа date.

```
int rjulmdy(date d, short mdy[3]);
```

Эта функция получает дату `d` и указатель на 3 коротких целочисленных значения `mdy`. Имя переменной указывает на порядок значений: в `mdy[0]` записывается номер месяца, в `mdy[1]` — номер дня, а в `mdy[2]` — год.

В текущем состоянии эта функция всегда возвращает 0.

Внутри эта функция вызывает `PGTYPESdate_julmdy`.

`rdefmtdate`

Преобразует символьную строку в значение типа date по маске формата.

```
int rdefmtdate(date *d, char *fmt, char *str);
```

Эта функция принимает указатель на переменную типа date (`d`), в которую будет помещён результат операции, маску формата для разбора даты (`fmt`) и строку C char*, содержащую текстовое представление даты (`str`). Ожидается, что текстовое представление будет соответствовать маске формата. Однако это соответствие не обязательно должно быть точным. Данная функция анализирует только порядок элементов и ищет в нём подстроки `yy` или `yyyy`, обозначающие позицию года, подстроку `mm`, обозначающую позицию месяца, и `dd`, обозначающую позицию дня.

Эта функция возвращает следующие значения:

- 0 — Функция выполнена успешно.
- `ECPG_INFORMIX_ENOSHORTDATE` — Дата не содержит разделителей между днём, месяцем и годом. С таким форматом входная строка должна быть длиной ровно 6 или 8 байт, но это не так.
- `ECPG_INFORMIX_ENOTDMY` — Строка формата не определяет корректно последовательный порядок года, месяца и дня.
- `ECPG_INFORMIX_BAD_DAY` — Во входной строке отсутствует корректное указание дня.
- `ECPG_INFORMIX_BAD_MONTH` — Во входной строке отсутствует корректное указание месяца.
- `ECPG_INFORMIX_BAD_YEAR` — Во входной строке отсутствует корректное указание года.

В реализации этой функции вызывается `PGTYPESdate_defmt_asc`. Примеры вводимых строк приведены в таблице в её описании.

`rfmtdate`

Преобразует переменную типа date в текстовое представление по маске формата.

```
int rfmtdate(date d, char *fmt, char *str);
```

Эта функция принимает дату для преобразования (`d`), маску формата (`fmt`) и строку, в которую будет помещено текстовое представление даты (`str`).

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

Внутри эта функция вызывает `PGTYPESdate_fmt_asc`, примеры форматов можно найти в её описании.

`rmdyjul`

Образует значение даты из массива 3 коротких целых, задающих день, месяц и год даты.

```
int rmdyjul(short mdy[3], date *d);
```

Эта функция получает в первом аргументе массив из 3 коротких целых (`mdy`), а во втором указатель на переменную типа date, в которую будет помещён результат операции.

В настоящее время эта функция всегда возвращает 0.

В реализации этой функции вызывается `PGTYPESdate_mdyjul`.

`rdayofweek`

Возвращает число, представляющее день недели для заданной даты.

```
int rdayofweek(date d);
```

Эта функция принимает в единственном аргументе переменную `d` типа date и возвращает целое число, выражающее день недели для этой даты.

- 0 — Воскресенье
- 1 — Понедельник
- 2 — Вторник
- 3 — Среда
- 4 — Четверг
- 5 — Пятница
- 6 — Суббота

В реализации этой функции вызывается PGTYPESdate_dayofweek.

dtcurrent

Получает текущее время.

```
void dtcurrent(timestamp *ts);
```

Эта функция получает текущее время и сохраняет его в переменной типа timestamp, на которую указывает ts.

dtcvasc

Разбирает время из текстового представления в переменную типа timestamp.

```
int dtcvasc(char *str, timestamp *ts);
```

Эта функция получает строку (str), которую нужно разобрать, и указатель на переменную типа timestamp, в которую будет помещён результат операции (ts).

Эта функция возвращает 0 в случае успеха, а в случае ошибки — отрицательное значение.

Внутри эта функция вызывает PGTYPEStimestamp_from_asc. Примеры вводимых строк приведены в таблице в её описании.

dtcvfmtasc

Разбирает время из текстового представления в переменную типа timestamp по маске формата.

```
dtcvfmtasc(char *inbuf, char *fmtstr, timestamp *dtvalue)
```

Эта функция получает строку (inbuf), которую нужно разобрать, маску формата (fmtstr) и указатель на переменную timestamp, в которой будет содержаться результат операции (dtvalue).

В реализации этой функции используется PGTYPEStimestamp_defmt_asc. Список допустимых кодов формата приведён в её описании.

Эта функция возвращает 0 в случае успеха, а в случае ошибки — отрицательное значение.

dtsub

Вычитает одно значение времени из другого и возвращает переменную типа interval.

```
int dtsub(timestamp *ts1, timestamp *ts2, interval *iv);
```

Эта функция вычитает значение timestamp, на которое указывает ts2, из значения timestamp, на которое указывает ts1, и сохраняет результат в переменной типа interval, на которую указывает iv.

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

dttoasc

Преобразует переменную типа timestamp в строку C char*.

```
int dttoasc(timestamp *ts, char *output);
```

Эта функция получает указатель (`ts`) на переменную типа timestamp, которую нужно преобразовать, и строку (`output`) для сохранения результата операции. Она преобразует `ts` в текстовое представление согласно стандарту SQL, то есть по маске `YYYY-MM-DD HH:MM:SS`.

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

`dttofmtasc`

Преобразует переменную типа timestamp в строку C char* по маске формата.

```
int dttofmtasc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

Эта функция получает в первом аргументе (`ts`) указатель на переменную типа timestamp, а в последующих указатель на буфер вывода (`output`), максимальную длину строки, которую может принять буфер (`str_len`), и маску формата, с которой будет выполняться преобразование (`fmtstr`).

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

Внутри эта функция использует `PGTYPEStimestamp_fmt_asc`. Примеры допустимых масок формата можно найти в её описании.

`intoasc`

Преобразует переменную типа interval в строку C char*.

```
int intoasc(interval *i, char *str);
```

Эта функция получает указатель (`i`) на переменную типа interval, которую нужно преобразовать, и строку (`str`) для сохранения результата операции. Она преобразует `i` в текстовое представление согласно стандарту SQL, то есть по маске `YYYY-MM-DD HH:MM:SS`.

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

`rfmtlong`

Преобразует длинное целое в текстовое представление по маске формата.

```
int rfmtlong(long lng_val, char *fmt, char *outbuf);
```

Эта функция принимает значение типа long (`lng_val`), маску формата (`fmt`) и указатель на выходной буфер (`outbuf`). Она преобразует длинное целое в его текстовое представление согласно заданной маске формата.

Маску формата можно составить из следующих символов, определяющих формат:

- `*` (звёздочка) — если в данной позиции будет пусто, заполнить её звёздочкой.
- `&` (амперсанд) — если в данной позиции будет пусто, заполнить её нулём.
- `#` — заменить ведущие нули пробелами.
- `<` — выровнять число в строке по левой стороне.
- `,` (запятая) — сгруппировать числа, содержащие четыре и более цифр, в группы по три цифры через запятую.
- `.` (точка) — этот символ отделяет целую часть числа от дробной.
- `-` (минус) — с отрицательным числом должен выводиться знак минус.
- `+` (плюс) — с положительным числом должен выводиться знак плюс.
- `(` — это символ заменяет знак минус перед отрицательным числом. Сам знак минус выводиться не будет.
- `)` — этот символ заменяет минус и выводится после отрицательного числа.

- `$` — символ денежной суммы.

`rupshift`

Приводит строку к верхнему регистру.

```
void rupshift(char *str);
```

Эта функция получает указатель на строку и приводит в ней каждый символ в нижнем регистре к верхнему регистру.

`byleng`

Возвращает число символов в строке, не считая завершающих пробелов.

```
int byleng(char *str, int len);
```

Эта функция принимает в первом аргументе (`str`) строку фиксированной длины, а во втором (`len`) её длину. Она возвращает число значимых символов, то есть длину строки без завершающих пробелов.

`ldchar`

Копирует строку фиксированной длины в строку с завершающим нулём.

```
void ldchar(char *src, int len, char *dest);
```

Эта функция принимает строку фиксированной длины (`src`), которую нужно скопировать, её длину (`len`) и указатель на целевой буфер в памяти (`dest`). Учтите, что для буфера, на который указывает `dest`, необходимо выделить как минимум `len+1` байт. Данная функция копирует в новую область не больше `len` байт (меньше, если в исходной строке есть завершающие пробелы) и добавляет завершающий 0.

`rgetmsg`

```
int rgetmsg(int msgnum, char *s, int maxsize);
```

Эта функция определена, но не реализована на данный момент!

`rtypalign`

```
int rtypalign(int offset, int type);
```

Эта функция определена, но не реализована на данный момент!

`rtypmsize`

```
int rtypmsize(int type, int len);
```

Эта функция определена, но не реализована на данный момент!

`rtypwidth`

```
int rtypwidth(int sqltype, int sqllen);
```

Эта функция определена, но не реализована на данный момент!

`rsetnull`

Присваивает переменной NULL.

```
int rsetnull(int t, char *ptr);
```

Эта функция получает целое, определяющее тип переменной, и указатель на саму переменную, приведённый к указателю C char*.

Определены следующие типы:

- `CCHARTYPE` — для переменной типа `char` или `char*`

- `CSHORTTYPE` — для переменной типа `short int`
- `CINTTYPE` — для переменной типа `int`
- `CBOOLTYPE` — для переменной типа `boolean`
- `CFLOATTYPE` — для переменной типа `float`
- `CLONGTYPE` — для переменной типа `long`
- `CDOUBLETYPE` — для переменной типа `double`
- `CDECIMALTYPE` — для переменной типа `decimal`
- `CDATETYPE` — для переменной типа `date`
- `CDTIMETYPE` — для переменной типа `timestamp`

Примеры вызова этой функции:

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;

rsetnull(CCHARTYPE, (char *) c);
rsetnull(CSHORTTYPE, (char *) &s);
rsetnull(CINTTYPE, (char *) &i);
```

`risnull`

Проверяет содержимое переменной на NULL.

```
int risnull(int t, char *ptr);
```

Эта функция получает тип проверяемой переменной (`t`), а также указатель на неё (`ptr`). Заметьте, что этот указатель нужно привести к char*. Список возможных типов переменных приведён в описании функции `rsetnull`.

Примеры использования этой функции:

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;

risnull(CCHARTYPE, (char *) c);
risnull(CSHORTTYPE, (char *) &s);
risnull(CINTTYPE, (char *) &i);
```

33.15.5. Дополнительные константы

Заметьте, что все эти константы относятся к ошибкам и все они представлены отрицательными значениями. Из описаний различных констант вы также можете узнать, какими именно числами они представлены в текущей реализации. Однако полагаться на эти числа не следует. Тем не менее вы можете рассчитывать на то, что все эти значения будут отрицательными.

`ECPG_INFORMIX_NUM_OVERFLOW`

Функции возвращают это значение, если при вычислении происходит переполнение. Внутри оно представляется числом -1200 (определение Informix).

`ECPG_INFORMIX_NUM_UNDERFLOW`

Функции возвращают это значение, если при вычислении происходит антипереполнение. Внутри оно представляется числом -1201 (определение Informix).

`ECPG_INFORMIX_DIVIDE_ZERO`

Функции возвращают это значение при попытке деления на ноль. Внутри оно представляется числом -1202 (определение Informix).

`ECPG_INFORMIX_BAD_YEAR`

Функции возвращают это значение, если при разборе даты встретилось некорректное указание года. Внутри оно представляется числом -1204 (определение Informix).

`ECPG_INFORMIX_BAD_MONTH`

Функции возвращают это значение, если при разборе даты встретилось некорректное указание месяца. Внутри оно представляется числом -1205 (определение Informix).

`ECPG_INFORMIX_BAD_DAY`

Функции возвращают это значение, если при разборе даты встретилось некорректное указание дня. Внутри оно представляется числом -1206 (определение Informix).

`ECPG_INFORMIX_ENOSHORTDATE`

Функции возвращают это значение, если процедуре разбора даты требуется короткая запись даты, но строка даты имеет неподходящую длину. Внутри оно представляется числом -1209 (определение Informix).

`ECPG_INFORMIX_DATE_CONVERT`

Функции возвращают это значение, если при форматировании даты происходит ошибка. Внутри оно представляется числом -1210 (определение Informix).

`ECPG_INFORMIX_OUT_OF_MEMORY`

Функции возвращают это значение, если им не хватает памяти для выполнения операций. Внутри оно представляется числом -1211 (определение Informix).

`ECPG_INFORMIX_ENOTDMY`

Функции возвращают это значение, если процедура разбора должна была получить маску формата (например, `mmddyy`), но не все поля были записаны правильно. Внутри оно представляется числом -1212 (определение Informix).

`ECPG_INFORMIX_BAD_NUMERIC`

Функции возвращают это значение, если процедура разбора не может получить числовое значение из текстового представления, потому что оно некорректно, либо если процедура вычисления не может произвести операцию с числовыми переменными из-за недопустимого значения минимум одной из этих переменных. Внутри оно представляется числом -1213 (определение Informix).

`ECPG_INFORMIX_BAD_EXPONENT`

Функции возвращают это значение, если процедура разбора не может воспринять экспоненту в числе. Внутри оно представляется числом -1216 (определение Informix).

`ECPG_INFORMIX_BAD_DATE`

Функции возвращают это значение, если процедура разбора не может разобрать дату. Внутри оно представляется числом -1218 (определение Informix).

`ECPG_INFORMIX_EXTRA_CHARS`

Функции возвращают это значение, если процедуре разбора передаются посторонние символы, которая она не может разобрать. Внутри оно представляется числом -1264 (определение Informix).

33.16. Внутреннее устройство

В этом разделе рассказывается, как препроцессор ECPG устроен внутри. Эта информация может оказаться полезной для пользователей, желающих понять, как использовать ECPG.

Первые четыре строки, которые `ecpg` записывает в вывод, фиксированы. Первые две строки содержат комментарии, а следующие две директивы включения, подключающие интерфейс к библиотеке. Затем препроцессор прочитывает файл и продолжает запись в вывод. Обычно он просто печатает всё в устройство вывода.

Встречая команду `EXEC SQL`, он вмешивается и изменяет её. Данная команда начинается со слов `EXEC SQL` и заканчивается знаком `;`. Всё между ними воспринимается как оператор SQL и разбирается для подстановки переменных.

Подстановка переменных имеет место, когда символ начинается с двоеточия (`:`). ECPG будет искать переменную с таким именем среди переменных, ранее объявленных в секции `EXEC SQL DECLARE`.

Самая важная функция в библиотеке — `ECPGdo`, которая осуществляет выполнение большинства команд. Она принимает переменное число аргументов (это число легко может достигать 50, и мы надеемся, что это не приведёт к проблемам ни на какой платформе).

Ей передаются следующие аргументы:

Номер строки

Номер исходной строки; используется только в сообщениях об ошибках.

Строка

Команда SQL, которая должна быть выполнена. На её содержимое влияют входные переменные, то есть переменные, добавленные в команду, но неизвестные во время компиляции. Места, в которые должны вставляться переменные, обозначаются знаками `?`.

Входные переменные

Для каждой входной переменной формируются десять аргументов. (См. ниже.)

`ECPGt_EOIT`

Перечисление (`enum`), показывающее, что больше входных переменных нет.

Выходные переменные

Для каждой входной переменной формируются десять аргументов. (См. ниже.) Эти переменные заполняются данной функцией.

`ECPGt_EORT`

Перечисление (`enum`), показывающее, что больше выходных переменных нет.

Для каждой переменной, включённой в команду SQL, эта функция принимает десять аргументов:

1. Тип в виде специального символа.

2. Указатель на значение или указатель на указатель.

3. Размер переменной, если она имеет тип `char` или `varchar`.

4. Число элементов в массиве (при выборке данных в массив).

5. Смещение следующего элемента в массиве (при выборке данных в массив).

6. Тип переменной-индикатора в виде специального символа.

7. Указатель на переменную-индикатор.

8. 0

9. Число элементов в массиве индикаторов (при выборке данных в массив).

10Смещение следующего элемента в массиве индикаторов (при выборке данных в массив).

Заметьте, что не все команды SQL обрабатываются таким образом. Например, команда открытия курсора вида:

```
EXEC SQL OPEN курсор;
```

не копируется в вывод. Вместо этого в позиции команды `OPEN` применяется команда `DECLARE` этого курсора, так как на самом деле курсор открывает она.

Ниже показан полный пример, демонстрирующий результат обработки препроцессором файла `foo.pgc` (детали могут меняться от версии к версии препроцессора):

```
EXEC SQL BEGIN DECLARE SECTION;
int index;
int result;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT res INTO :result FROM mytable WHERE index = :index;
```

преобразуется в:

```
/* Processed by ecpg (2.6.0) */
/* These two include files are added by the preprocessor */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* exec sql begin declare section */

#line 1 "foo.pgc"

 int index;
 int result;
/* exec sql end declare section */
...
ECPGdo(__LINE__, NULL, "SELECT res FROM mytable WHERE index = ?     ",
        ECPGt_int,&(index),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(result),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
#line 147 "foo.pgc"
```

(Отступы здесь добавлены для читаемости, препроцессор их не вставляет.)

Глава 34. Информационная схема

Информационная схема состоит из набора представлений, содержащих информацию об объектах, определённых в текущей базе данных. Информационная схема описана в стандарте SQL и поэтому можно рассчитывать на её переносимость и стабильность — в отличие от системных каталогов, которые привязаны к Postgres Pro, и моделируются, отталкиваясь от реализации. Представления информационной схемы, однако, не содержат информацию о функциях, присущих исключительно Postgres Pro; чтобы получить информацию о них, необходимо обратиться к системным каталогам или другим специфическим представлениям Postgres Pro.

> **Примечание**
>
> Когда из базы данных запрашивается информация об ограничениях, возможна ситуация, когда соответствующий стандарту запрос, который должен возвращать одну строку, возвращает несколько. Это связано с тем, что стандарт SQL требует, чтобы имена ограничений были уникальными в схеме, но в Postgres Pro такого требования нет. Имена ограничений, которые Postgres Pro генерирует автоматически, не должны дублироваться в одной схеме, но сами пользователи могут назначить подобные дублирующиеся имена.
>
> Эта проблема может проявиться при обращении к таким представлениям информационной схемы, как `check_constraint_routine_usage`, `check_constraints`, `domain_constraints` и `referential_constraints`. В некоторых других представлениях она могла бы тоже иметь место, но они содержат имя таблицы, помогающее различить дублирующиеся строки, например: `constraint_column_usage`, `constraint_table_usage`, `table_constraints`.

34.1. Схема

Информационная схема сама по себе — это схема с именем `information_schema`. Данная схема автоматически доступна во всех базах данных. Владельцем этой схемы является начальный пользователь баз данных в кластере, и этот пользователь, естественно, имеет все права в ней, включая возможность её удалить (хотя достигаемая при этом экономия пространства минимальна).

По умолчанию информационная схема отсутствует в пути поиска схем, так что ко всем объектам в ней нужно обращаться по полным именам. Так как имена некоторых объектов в информационной схеме довольно распространённые и могут встречаться и в пользовательских приложениях, будьте осторожны, добавляя информационную схему в путь поиска.

34.2. Типы данных

Столбцы в представлениях информационной схемы имеют специальные типы данных, определённые в информационной схеме. Они определены как простые домены поверх обычных встроенных типов. Задействовать эти типы вне информационной схемы не следует, но тем не менее приложения, выбирающие данные из информационной схеме должны быть готовы работать с ними.

Это следующие типы:

`cardinal_number`

Неотрицательное целое.

`character_data`

Строка символов (без определённого ограничения по длине).

`sql_identifier`

Строка символов. Этот тип применяется для идентификаторов SQL, тогда как тип `character_data` для всех остальных видов текстовых данных.

time_stamp

Домен на базе типа timestamp with time zone

yes_or_no

Домен символьной строки, который принимает либо YES, либо NO. Этот домен применяется для представления булевых данных (истина/ложь, true/false) в информационной схеме. (Информационная схема была введена до появления в стандарте SQL типа boolean, поэтому данный домен необходим для сохранения обратной совместимости информационной схемы.)

Все столбцы в информационной схеме имеют один из этих пяти типов.

34.3. information_schema_catalog_name

Таблица information_schema_catalog_name всегда содержит одну строку и один столбец с именем текущей базы данных (текущий каталог, в терминологии SQL).

Таблица 34.1. Столбцы information_schema_catalog_name

Имя	Тип данных	Описание
catalog_name	sql_identifier	Имя базы данных, содержащей эту информационную схему

34.4. administrable_role_authorizations

Представление administrable_role_authorizations описывает все роли, для которых текущий пользователь является администратором.

Таблица 34.2. Столбцы administrable_role_authorizations

Имя	Тип данных	Описание
grantee	sql_identifier	Имя роли, которой было разрешено участие в целевой роли (может быть текущий пользователь, либо другая роль, в случае вложенного членства)
role_name	sql_identifier	Имя целевой роли
is_grantable	yes_or_no	Всегда YES

34.5. applicable_roles

Представление applicable_roles описывает все роли, права которых может использовать текущий пользователь. Это означает, что существует некоторая цепочка ролей от текущего пользователя к целевой роли. Роль самого пользователя также считается применимой. Набор применимых ролей обычно используется для проверки разрешений.

Таблица 34.3. Столбцы applicable_roles

Имя	Тип данных	Описание
grantee	sql_identifier	Имя роли, которой было разрешено участие в целевой роли (может быть текущий пользователь, либо другая роль, в случае вложенного членства)
role_name	sql_identifier	Имя целевой роли
is_grantable	yes_or_no	YES, если субъект является администратором для этой роли, или NO в противном случае

34.6. attributes

Представление attributes содержит информацию об атрибутах составных типов данных, определённых в базе. (Заметьте, что представление не даёт информацию о столбцах таблицы, которые иногда называются атрибутами в контекстах Postgres Pro.) В нём показываются только те атрибуты, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторое право для использования типа).

Таблица 34.4. Столбцы attributes

Имя	Тип данных	Описание
udt_catalog	sql_identifier	Имя базы данных, содержащей тип данных (всегда текущая база)
udt_schema	sql_identifier	Имя схемы, содержащей тип данных
udt_name	sql_identifier	Имя типа данных
attribute_name	sql_identifier	Имя атрибута
ordinal_position	cardinal_number	Порядковый номер атрибута внутри типа данных (нумерация начинается с 1)
attribute_default	character_data	Выражение по умолчанию для атрибута
is_nullable	yes_or_no	YES, если атрибут может содержать NULL, или NO, если он не принимает NULL
data_type	character_data	Тип данных атрибута, если это встроенный тип, либо ARRAY, если это массив (в этом случае обратитесь к представлению element_types), иначе — USER-DEFINED (в этом случае тип определяется в attribute_udt_name и связанных столбцах).
character_maximum_length	cardinal_number	Если в data_type указан тип текстовой или битовой строки, это поле задаёт её объявленную максимальную длину; NULL для всех других типов данных, либо если максимальная длина не объявлена.
character_octet_length	cardinal_number	Если в data_type указан тип символьной строки, это поле задаёт её максимально возможный размер в октетах (байтах); NULL для всех других типов данных. Максимальный размер в октетах зависит от объявленной максимальной длины в символах (см. выше) и от кодировки сервера.

Имя	Тип данных	Описание
character_set_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
character_set_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
character_set_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
collation_catalog	sql_identifier	Имя базы данных, содержащей правило сортировки атрибута (это всегда текущая база), либо NULL, если это правило по умолчанию или тип данных атрибута несортируемый
collation_schema	sql_identifier	Имя схемы, содержащей правило сортировки атрибута, либо NULL, если это правило по умолчанию или тип данных атрибута несортируемый
collation_name	sql_identifier	Имя правила сортировки атрибута, либо NULL, если это правило по умолчанию или атрибут несортируемый
numeric_precision	cardinal_number	Если в data_type указан числовой тип, этот столбец содержит точность (объявленную или неявную) типа для этого атрибута. Точность определяет число значащих цифр. Она может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу numeric_ precision_radix . Для всех других типов данных этот столбец содержит NULL.
numeric_precision_radix	cardinal_number	Если в data_type указан числовой тип, в этом столбце определяется, по какому основанию задаются значения в столбцах numeric_precision и numeric_scale . Возможные варианты: 2 или 10. Для всех других типов данных этот столбец содержит NULL.
numeric_scale	cardinal_number	Если в data_type указан точный числовой тип, этот столбец содержит масштаб (объявленный или неявный) типа для этого атрибута. Масштаб определяет число

Имя	Тип данных	Описание
		значащих цифр справа от десятичной точки. Он может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу numeric_precision_radix . Для всех других типов данных этот столбец содержит NULL.
datetime_precision	cardinal_number	Если в data_type указан тип даты, времени, отметки времени или интервала, этот столбец содержит точность (объявленную или неявную) в долях секунды типа этого атрибута, то есть число десятичных цифр, сохраняемых после десятичной точки в значении секунд. Для всех других типов данных этот столбец содержит NULL.
interval_type	character_data	Если в data_type указан тип интервала, этот столбец определяет, какие поля принимает интервал в этом атрибуте, например: YEAR TO MONTH, DAY TO SECOND и т. д. Если ограничения для полей не заданы (то есть, интервал принимает все поля), и для любых других типов данных это поле содержит NULL.
interval_precision	cardinal_number	Относится к функциональности, отсутствующей в Postgres Pro (см. поле datetime_precision , определяющее точность в долях секунды для типов интервалов)
attribute_udt_catalog	sql_identifier	Имя базы данных, в которой определён тип данных атрибута (всегда текущая база)
attribute_udt_schema	sql_identifier	Имя схемы, в которой определён тип данных атрибута
attribute_udt_name	sql_identifier	Имя типа данных атрибута
scope_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
scope_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro

Имя	Тип данных	Описание
scope_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
maximum_cardinality	cardinal_number	Всегда NULL, так как массивы имеют неограниченную максимальную ёмкость в Postgres Pro
dtd_identifier	sql_identifier	Идентификатор дескриптора типа данных столбца, уникальный среди всех дескрипторов типов, относящихся к таблице. Он в основном полезен для соединения с другими экземплярами таких идентификаторов. (Конкретный формат идентификатора не определён и не гарантируется, что он останется неизменным в будущих версиях.)
is_derived_reference_attribute	yes_or_no	Относится к функциональности, отсутствующей в Postgres Pro

Обратитесь также к описанию Раздел 34.16, представлению с похожей структурой, за дополнительной информацией о некоторых столбцах.

34.7. character_sets

Представление character_sets описывает наборы символов, доступные в текущей базе данных. Так как Postgres Pro не поддерживает несколько наборов символов в одной базе данных, этот набор показывает только один набор, соответствующий кодировке базы.

Примите к сведению следующие термины, принятые в стандарте SQL:

совокупность символов

Абстрактная коллекция символов, например UNICODE, UCS или LATIN1. Не существует в виде SQL-объекта, но показывается в этом представлении.

форма кодировки символов

Кодировка некоторой совокупности символов. Для большинства устаревших совокупностей используется только одна кодировка, так что эта кодировка не имеет отдельного имени (например, LATIN1 — форма кодировки, применимая к совокупности LATIN1). Но например, Unicode имеет формы кодировки UTF8, UTF16 и т. д. (не все они поддерживаются в Postgres Pro). Формы кодировки не существуют в виде SQL-объектов, но показываются в этом представлении.

набор символов

Именованный SQL-объект, определяющий совокупность и кодировку символов, а также правило сортировки по умолчанию. Предопределённый набор символов обычно называется так же, как форма кодировки, но пользователи могут определить другие имена. Например, набору символов UTF8 обычно соответствует совокупность символов UCS, форма кодировки UTF8 и некоторое правило сортировки по умолчанию.

Вы можете считать, что «кодировка» в Postgres Pro определяет набор или форму кодировки символов. Она имеет такое же имя и может быть только одной в определённой базе.

Таблица 34.5. Столбцы character_sets

Имя	Тип данных	Описание
character_set_catalog	sql_identifier	Наборы символов в настоящее время не представлены в виде объектов схемы, так что этот столбец содержит NULL.
character_set_schema	sql_identifier	Наборы символов в настоящее время не представлены в виде объектов схемы, так что этот столбец содержит NULL.
character_set_name	sql_identifier	Имя набора символов, в настоящее время в качестве этого имени показывается имя кодировки базы данных
character_repertoire	sql_identifier	Совокупность символов — UCS для кодировки UTF8, либо просто имя кодировки
form_of_use	sql_identifier	Форма кодировки символов, то же, что и кодировка базы данных
default_collate_catalog	sql_identifier	Имя базы данных, содержащей правило сортировки по умолчанию (всегда текущая база, если это правило установлено)
default_collate_schema	sql_identifier	Имя схемы, содержащей правило сортировки по умолчанию
default_collate_name	sql_identifier	Имя правила сортировки по умолчанию. Правилом сортировки по умолчанию считается правило, соответствующее параметрам COLLATE и CTYPE текущей базы данных. Если такого правила нет, данный столбец и связанные столбцы схемы и каталога содержат NULL.

34.8. check_constraint_routine_usage

Представление check_constraint_routine_usage описывает подпрограммы (функции и процедуры), участвующие в проверках ограничений. В нём показываются только те подпрограммы, которые принадлежат текущей активной роли.

Таблица 34.6. Столбцы check_constraint_routine_usage

Имя	Тип данных	Описание
constraint_catalog	sql_identifier	Имя базы данных, содержащей ограничение (всегда текущая база)
constraint_schema	sql_identifier	Имя схемы, содержащей ограничение

Имя	Тип данных	Описание
constraint_name	sql_identifier	Имя ограничения
specific_catalog	sql_identifier	Имя базы данных, содержащей функцию (всегда текущая база)
specific_schema	sql_identifier	Имя схемы, содержащей функцию
specific_name	sql_identifier	«Однозначное имя» функции. Подробнее оно описано в Разделе 34.40.

34.9. check_constraints

Представление check_constraints показывает все ограничения-проверки, либо определённые для таблицы или домена, либо принадлежащие текущей активной роли. (Владелец таблицы или домена является владельцем ограничения.)

Таблица 34.7. Столбцы check_constraints

Имя	Тип данных	Описание
constraint_catalog	sql_identifier	Имя базы данных, содержащей ограничение (всегда текущая база)
constraint_schema	sql_identifier	Имя схемы, содержащей ограничение
constraint_name	sql_identifier	Имя ограничения
check_clause	character_data	Выражение проверки для ограничения

34.10. collations

Представление collations показывает правила сортировки, доступные в текущей базе данных.

Таблица 34.8. Столбцы collations

Имя	Тип данных	Описание
collation_catalog	sql_identifier	Имя базы данных, содержащей правило сортировки (всегда текущая база)
collation_schema	sql_identifier	Имя схемы, содержащей правило сортировки
collation_name	sql_identifier	Имя правила сортировки
pad_attribute	character_data	Всегда NO PAD (Альтернативный вариант PAD SPACE в Postgres Pro не поддерживается.)

34.11. collation_character_set_applicability

Представление collation_character_set_applicability показывает, к каким наборам символов применимы доступные правила сортировки. В Postgres Pro в базе данных может быть только один набор символов (см. описание в Разделе 34.7), так что это представление даёт не так много полезной информации.

Таблица 34.9. Столбцы `collation_character_set_applicability`

Имя	Тип данных	Описание
`collation_catalog`	`sql_identifier`	Имя базы данных, содержащей правило сортировки (всегда текущая база)
`collation_schema`	`sql_identifier`	Имя схемы, содержащей правило сортировки
`collation_name`	`sql_identifier`	Имя правила сортировки
`character_set_catalog`	`sql_identifier`	Наборы символов в настоящее время не представлены в виде объектов схемы, так что этот столбец содержит NULL
`character_set_schema`	`sql_identifier`	Наборы символов в настоящее время не представлены в виде объектов схемы, так что этот столбец содержит NULL
`character_set_name`	`sql_identifier`	Имя набора символов

34.12. `column_domain_usage`

Представление `column_domain_usage` описывает все столбцы (таблиц или представлений), которые используют какие-либо домены, определённые в базе данных и принадлежащие текущей активной роли.

Таблица 34.10. Столбцы `column_domain_usage`

Имя	Тип данных	Описание
`domain_catalog`	`sql_identifier`	Имя базы данных, содержащей домен (всегда текущая база)
`domain_schema`	`sql_identifier`	Имя схемы, содержащей домен
`domain_name`	`sql_identifier`	Имя домена
`table_catalog`	`sql_identifier`	Имя базы данных, содержащей таблицу (всегда текущая база)
`table_schema`	`sql_identifier`	Имя схемы, содержащей таблицу
`table_name`	`sql_identifier`	Имя таблицы
`column_name`	`sql_identifier`	Имя столбца

34.13. `column_options`

Представление `column_options` показывает все параметры, определённые для столбцов сторонней таблицы в текущей базе данных. В нём отражаются только те столбцы сторонних таблиц, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица 34.11. Столбцы `column_options`

Имя	Тип данных	Описание
`table_catalog`	`sql_identifier`	Имя базы данных, содержащей стороннюю таблицу (всегда текущая база)
`table_schema`	`sql_identifier`	Имя схемы, содержащей стороннюю таблицу

Имя	Тип данных	Описание
table_name	sql_identifier	Имя сторонней таблицы
column_name	sql_identifier	Имя столбца
option_name	sql_identifier	Имя параметра
option_value	character_data	Значение параметра

34.14. column_privileges

Представление column_privileges описывает все права, назначенные текущей активной роли или текущей активной ролью для столбцов. Оно содержит отдельную строку для каждой комбинации столбца, праводателя и правообладателя.

Если право даётся для всей таблицы, оно будет показываться как право для каждого столбца, но только для типов прав, применимых к столбцам: SELECT, INSERT, UPDATE, REFERENCES.

Таблица 34.12. Столбцы column_privileges

Имя	Тип данных	Описание
grantor	sql_identifier	Имя роли, давшей право (праводатель)
grantee	sql_identifier	Имя роли, которой было дано право (правообладатель)
table_catalog	sql_identifier	Имя базы данных, содержащей таблицу с этим столбцом (всегда текущая база)
table_schema	sql_identifier	Имя схемы, содержащей таблицу с этим столбцом
table_name	sql_identifier	Имя таблицы с этим столбцом
column_name	sql_identifier	Имя столбца
privilege_type	character_data	Тип права: SELECT, INSERT, UPDATE или REFERENCES
is_grantable	yes_or_no	YES, если право может передаваться, или NO в противном случае

34.15. column_udt_usage

Представление column_udt_usage описывает все столбцы, которые используют типы данных, принадлежащие текущей активной роли. Заметьте, что в Postgres Pro встроенные типы данных не отличаются от определённых пользователем, так что в этом представлении выводятся и они. За подробностями обратитесь к Разделу 34.16.

Таблица 34.13. Столбцы column_udt_usage

Имя	Тип данных	Описание
udt_catalog	sql_identifier	Имя базы данных, в которой определён тип (если применимо, нижележащий тип домена) столбца (всегда текущая база)
udt_schema	sql_identifier	Имя схемы, в которой определён тип (если применимо, нижележащий тип

Имя	Тип данных	Описание
		домена) столбца (всегда текущая база)
udt_name	sql_identifier	Имя типа данных столбца (если применимо, нижележащий тип домена)
table_catalog	sql_identifier	Имя базы данных, содержащей таблицу (всегда текущая база)
table_schema	sql_identifier	Имя схемы, содержащей таблицу
table_name	sql_identifier	Имя таблицы
column_name	sql_identifier	Имя столбца

34.16. columns

Представление columns содержит информацию обо всех столбцах таблиц (или столбцах представлений) в базе данных. Системные столбцы (oid и т. д.) в нём не отображаются. В нём показываются только те столбцы, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 34.14. Столбцы columns

Имя	Тип данных	Описание
table_catalog	sql_identifier	Имя базы данных, содержащей таблицу (всегда текущая база)
table_schema	sql_identifier	Имя схемы, содержащей таблицу
table_name	sql_identifier	Имя таблицы
column_name	sql_identifier	Имя столбца
ordinal_position	cardinal_number	Порядковый номер столбца в таблице (нумерация начинается с 1)
column_default	character_data	Выражение по умолчанию для столбца
is_nullable	yes_or_no	YES, если столбец может содержать NULL, или NO, если он не принимает NULL. Не будет принимать NULL столбец с ограничением NOT NULL, но возможны и другие варианты.
data_type	character_data	Тип данных столбца, если это встроенный тип, либо ARRAY, если это массив (в этом случае обратитесь к представлению element_types), иначе — USER-DEFINED (в этом случае тип определяется в udt_name и связанных столбцах). Если столбец основан на домене, данный столбец показывает нижележащий тип домена (а сам домен показывается

Имя	Тип данных	Описание
		в domain_name и связанных столбцах).
character_maximum_length	cardinal_number	Если в data_type указан тип текстовой или битовой строки, это поле задаёт её объявленную максимальную длину; NULL для всех других типов данных, либо если максимальная длина не объявлена.
character_octet_length	cardinal_number	Если в data_type указан тип символьной строки, это поле задаёт её максимально возможный размер в октетах (байтах); NULL для всех других типов данных. Максимальный размер в октетах зависит от объявленной максимальной длины в символах (см. выше) и от кодировки сервера.
numeric_precision	cardinal_number	Если в data_type указан числовой тип, этот столбец содержит точность (объявленную или неявную) типа для целевого столбца. Точность определяет число значащих цифр. Она может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу numeric_ precision_radix . Для всех других типов данных этот столбец содержит NULL.
numeric_precision_radix	cardinal_number	Если в data_type указан числовой тип, в этом столбце определяется, по какому основанию задаются значения в столбцах numeric_precision и numeric_scale . Возможные варианты: 2 или 10. Для всех других типов данных этот столбец содержит NULL.
numeric_scale	cardinal_number	Если в data_type указан точный числовой тип, этот столбец содержит масштаб (объявленный или неявный) типа для целевого столбца. Масштаб определяет число значащих цифр справа от десятичной точки. Он может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу numeric_

Имя	Тип данных	Описание
		precision_radix . Для всех других типов данных этот столбец содержит NULL.
datetime_precision	cardinal_number	Если в data_type указан тип даты, времени, отметки времени или интервала, этот столбец содержит точность (объявленную или неявную) в долях секунды типа для целевого столбца, то есть число десятичных цифр, сохраняемых после десятичной точки в значении секунд. Для всех других типов данных этот столбец содержит NULL.
interval_type	character_data	Если в data_type указан тип интервала, этот столбец определяет, какие поля принимает интервал в целевом столбце, например: YEAR TO MONTH, DAY TO SECOND и т. д. Если ограничения для полей не заданы (то есть, интервал принимает все поля), и для любых других типов данных это поле содержит NULL.
interval_precision	cardinal_number	Относится к функциональности, отсутствующей в Postgres Pro (см. поле datetime_precision , определяющее точность в долях секунды для типов интервалов)
character_set_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
character_set_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
character_set_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
collation_catalog	sql_identifier	Имя базы данных, содержащей правило сортировки столбца (это всегда текущая база), либо NULL, если это правило по умолчанию или тип данных столбца несортируемый
collation_schema	sql_identifier	Имя схемы, содержащей правило сортировки столбца, либо NULL, если это правило по умолчанию или тип данных столбца несортируемый

Имя	Тип данных	Описание
collation_name	sql_identifier	Имя правила сортировки столбца, либо NULL, если это правило по умолчанию или тип данных столбца несортируемый
domain_catalog	sql_identifier	Если целевой столбец имеет тип домена, этот столбец содержит имя базы данных, в которой определён домен (всегда текущая база), иначе — NULL.
domain_schema	sql_identifier	Если целевой столбец имеет тип домена, этот столбец содержит имя схемы, в которой определён домен, иначе — NULL.
domain_name	sql_identifier	Если целевой столбец имеет тип домена, этот столбец содержит имя домена, иначе — NULL.
udt_catalog	sql_identifier	Имя базы данных, в которой определён тип (если применимо, нижележащий тип домена) столбца (всегда текущая база)
udt_schema	sql_identifier	Имя схемы, в которой определён тип (если применимо, нижележащий тип домена) столбца (всегда текущая база)
udt_name	sql_identifier	Имя типа данных столбца (если применимо, нижележащий тип домена)
scope_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
scope_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
scope_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
maximum_cardinality	cardinal_number	Всегда NULL, так как массивы имеют неограниченную максимальную ёмкость в Postgres Pro
dtd_identifier	sql_identifier	Идентификатор дескриптора типа данных столбца, уникальный среди всех дескрипторов типов, относящихся к таблице. Он в основном полезен для соединения с другими экземплярами таких идентификаторов. (Конкретный

Имя	Тип данных	Описание
		формат идентификатора не определён и не гарантируется, что он останется неизменным в будущих версиях.)
is_self_referencing	yes_or_no	Относится к функциональности, отсутствующей в Postgres Pro
is_identity	yes_or_no	Если целевой столбец является столбцом идентификации, значение YES, иначе — NO.
identity_generation	character_data	Если целевой столбец является столбцом идентификации, значение ALWAYS или BY DEFAULT, отражающее определение столбца.
identity_start	character_data	Если целевой столбец является столбцом идентификации, начальное значение внутренней последовательности, иначе — NULL.
identity_increment	character_data	Если целевой столбец является столбцом идентификации, шаг внутренней последовательности, иначе — NULL.
identity_maximum	character_data	Если целевой столбец является столбцом идентификации, максимальное значение внутренней последовательности, иначе — NULL.
identity_minimum	character_data	Если целевой столбец является столбцом идентификации, минимальное значение внутренней последовательности, иначе — NULL.
identity_cycle	yes_or_no	Если целевой столбец является столбцом идентификации, YES показывает, что внутренняя последовательность зацикливается, NO — не зацикливается, иначе — NULL.
is_generated	character_data	Относится к функциональности, отсутствующей в Postgres Pro
generation_expression	character_data	Относится к функциональности, отсутствующей в Postgres Pro
is_updatable	yes_or_no	YES, если столбец допускает изменение, или NO в противном

Имя	Тип данных	Описание
		случае (столбцы в базовых таблицах всегда изменяемые, но в представлениях — не обязательно)

Так как типы данных могут определяться в SQL множеством способов и Postgres Pro добавляет дополнительные варианты, представление типов в информационной схеме может быть довольно сложным. Столбец `data_type` предназначен для идентификации нижележащего встроенного типа столбца. В Postgres Pro это означает, что данный тип определён в схеме системного каталога `pg_catalog`. Этот столбец может быть полезным, если приложение способно особым образом воспринимать встроенные типы (например, форматировать числовые типы по-другому или задействовать данные в столбцах точности). Столбцы `udt_name`, `udt_schema` и `udt_catalog` всегда указывают на нижележащий тип данных столбца, даже если столбец основан на домене. (Так как в Postgres Pro встроенные типы не отличаются от определённых пользователем, в этом представлении выводятся и они. Это расширение стандарта SQL.) Эти столбцы должны учитываться, когда приложению нужно обрабатывать данные в зависимости от типа, так как в этом случае не важно, основан ли столбец на домене. Если столбец основан на домене, на него указывают столбцы `domain_name`, `domain_schema` и `domain_catalog`. Если вы хотите связать столбцы с их типами данных и обработать домены как отдельные типы, вы можете записать `coalesce(domain_name, udt_name)` и т. п.

34.17. constraint_column_usage

Представление `constraint_column_usage` описывает все столбцы в текущей базе данных, связанные с некоторым ограничением. В нём показываются только столбцы таблиц, принадлежащих текущей активной роли. Для ограничений-проверок это представление содержит столбцы, задействованные в выражении проверки. Для ограничений внешнего ключа оно содержит столбцы, на которые ссылается внешний ключ, а для ограничений уникальности или первичного ключа — ограничиваемые столбцы.

Таблица 34.15. Столбцы `constraint_column_usage`

Имя	Тип данных	Описание
`table_catalog`	`sql_identifier`	Имя базы данных, содержащей таблицу со столбцом, задействованным в некотором ограничении (всегда текущая база)
`table_schema`	`sql_identifier`	Имя схемы, содержащей таблицу со столбцом, задействованным в некотором ограничении
`table_name`	`sql_identifier`	Имя таблицы со столбцом, задействованным в некотором ограничении
`column_name`	`sql_identifier`	Имя столбца, задействованного в некотором ограничении
`constraint_catalog`	`sql_identifier`	Имя базы данных, содержащей ограничение (всегда текущая база)
`constraint_schema`	`sql_identifier`	Имя схемы, содержащей ограничение
`constraint_name`	`sql_identifier`	Имя ограничения

34.18. constraint_table_usage

Представление constraint_table_usage описывает все таблицы в текущей базе данных, связанные с некоторым ограничением и принадлежащие текущей активной роли. (Это отличается от представления table_constraints, в котором показываются все ограничения таблиц с таблицами, для которых они определены.) Для ограничений внешнего ключа это представление показывает таблицу, на которую ссылается ограничение. Для ограничений уникальности или первичного ключа в этом представлении показывается таблица, которой принадлежит ограничение. Ограничения-проверки и ограничения NOT NULL в нём не отражаются.

Таблица 34.16. Столбцы constraint_table_usage

Имя	Тип данных	Описание
table_catalog	sql_identifier	Имя базы данных, которая содержит таблицу, задействованную некоторым ограничением (всегда текущая база)
table_schema	sql_identifier	Имя схемы, которая содержит таблицу, задействованную некоторым ограничением
table_name	sql_identifier	Имя таблицы, задействованной некоторым ограничением
constraint_catalog	sql_identifier	Имя базы данных, содержащей ограничение (всегда текущая база)
constraint_schema	sql_identifier	Имя схемы, содержащей ограничение
constraint_name	sql_identifier	Имя ограничения

34.19. data_type_privileges

Представление data_type_privileges описывает все дескрипторы типов данных, к которым имеет доступ текущий пользователь, являясь их владельцем или имея некоторые права для них. Дескриптор типа данных формируется, когда тип данных задействуется в определении столбца таблицы, домена или функции (в качестве типа параметра или результата), и хранит некоторую информацию о том, как этот тип используется в данном случае (например, объявленную максимальную длину, если это применимо). Каждому дескриптору типа данных назначается уникальный идентификатор, уникальный среди всех дескрипторов типов, назначаемых одному объекту (таблица, домен, функция). Это представление, вероятно, не полезно для приложений, но на его базе определены некоторые другие представления в информационной схеме.

Таблица 34.17. Столбцы data_type_privileges

Имя	Тип данных	Описание
object_catalog	sql_identifier	Имя базы данных, содержащей описываемый объект (всегда текущая база)
object_schema	sql_identifier	Имя схемы, содержащей описываемый объект
object_name	sql_identifier	Имя описываемого объекта
object_type	character_data	Тип описываемого объекта: TABLE (дескриптор типа данных относится к столбцу этой таблицы), DOMAIN (дескриптор

Имя	Тип данных	Описание
		типа данных относится к домену) или ROUTINE (дескриптор типа данных относится к типу данных параметра или результата функции).
dtd_identifier	sql_identifier	Идентификатор дескриптора типа данных, уникальный среди дескрипторов типов для этого же объекта.

34.20. domain_constraints

Представление domain_constraints показывает все ограничения, принадлежащие доменам, определённым в текущей базе данных. В нём отражаются только те домены, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 34.18. Столбцы domain_constraints

Имя	Тип данных	Описание
constraint_catalog	sql_identifier	Имя базы данных, содержащей ограничение (всегда текущая база)
constraint_schema	sql_identifier	Имя схемы, содержащей ограничение
constraint_name	sql_identifier	Имя ограничения
domain_catalog	sql_identifier	Имя базы данных, содержащей домен (всегда текущая база)
domain_schema	sql_identifier	Имя схемы, содержащей домен
domain_name	sql_identifier	Имя домена
is_deferrable	yes_or_no	YES, если ограничение откладываемое, или NO в противном случае
initially_deferred	yes_or_no	YES, если ограничение откладываемое и отложенное изначально, или NO в противном случае

34.21. domain_udt_usage

Представление domain_udt_usage описывает все домены, которые используют типы данных, принадлежащие текущей активной роли. Заметьте, что в Postgres Pro встроенные типы данных не отличаются от определённых пользователем, так что в этом представлении выводятся и они.

Таблица 34.19. Столбцы domain_udt_usage

Имя	Тип данных	Описание
udt_catalog	sql_identifier	Имя базы данных, в которой определён тип данных домена (всегда текущая база)
udt_schema	sql_identifier	Имя схемы, в которой определён тип данных домена
udt_name	sql_identifier	Имя типа данных домена

Имя	Тип данных	Описание
domain_catalog	sql_identifier	Имя базы данных, содержащей домен (всегда текущая база)
domain_schema	sql_identifier	Имя схемы, содержащей домен
domain_name	sql_identifier	Имя домена

34.22. domains

Представление domains показывает все домены, определённые в текущей базе данных. В нём показываются только те домены, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 34.20. Столбцы domains

Имя	Тип данных	Описание
domain_catalog	sql_identifier	Имя базы данных, содержащей домен (всегда текущая база)
domain_schema	sql_identifier	Имя схемы, содержащей домен
domain_name	sql_identifier	Имя домена
data_type	character_data	Тип данных домена, если это встроенный тип, либо ARRAY, если это массив (в этом случае обратитесь к представлению element_types), иначе — USER-DEFINED (в этом случае тип определяется в udt_name и связанных столбцах).
character_maximum_length	cardinal_number	Если домен имеет тип текстовой или битовой строки, это поле задаёт её объявленную максимальную длину; NULL для всех других типов данных, или если максимальная длина не объявлена.
character_octet_length	cardinal_number	Если домен имеет тип символьной строки, это поле задаёт её максимально возможный размер в октетах (байтах); NULL для всех других типов данных. Максимальный размер в октетах зависит от объявленной максимальной длины в символах (см. выше) и от кодировки сервера.
character_set_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
character_set_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
character_set_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro

Имя	Тип данных	Описание
collation_catalog	sql_identifier	Имя базы данных, содержащей правило сортировки домена (это всегда текущая база), либо NULL, если это правило по умолчанию или тип домена несортируемый
collation_schema	sql_identifier	Имя схемы, содержащей правило сортировки домена, либо NULL, если это правило по умолчанию или тип домена несортируемый
collation_name	sql_identifier	Имя правила сортировки домена, либо NULL, если это правило по умолчанию или тип домена несортируемый
numeric_precision	cardinal_number	Если домен имеет числовой тип, этот столбец содержит точность (объявленную или неявную) типа для этого домена. Точность определяет число значащих цифр. Она может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу numeric_precision_radix . Для всех других типов данных этот столбец содержит NULL.
numeric_precision_radix	cardinal_number	Если домен имеет числовой тип, в этом столбце определяется, по какому основанию задаются значения в столбцах numeric_precision и numeric_scale . Возможные варианты: 2 и 10. Для всех других типов данных этот столбец содержит NULL.
numeric_scale	cardinal_number	Если домен имеет точный числовой тип, этот столбец содержит масштаб (объявленный или неявный) типа для этого домена. Масштаб определяет число значащих цифр справа от десятичной точки. Он может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу numeric_precision_radix . Для всех других типов данных этот столбец содержит NULL.

Имя	Тип данных	Описание
datetime_precision	cardinal_number	Если в data_type указан тип даты, времени, отметки времени или интервала, этот столбец содержит точность (объявленную или неявную) в долях секунды типа для этого домена, то есть число десятичных цифр, сохраняемых после десятичной точки в значении секунд. Для всех других типов данных этот столбец содержит NULL.
interval_type	character_data	Если в data_type указан тип интервала, этот столбец определяет, какие поля принимает интервал в домене, например: YEAR TO MONTH, DAY TO SECOND и т. д. Если ограничения для полей не заданы (то есть, интервал принимает все поля), и для любых других типов данных это поле содержит NULL.
interval_precision	cardinal_number	Относится к функциональности, отсутствующей в Postgres Pro (см. поле datetime_precision , определяющее точность в долях секунды для типов интервалов)
domain_default	character_data	Выражение по умолчанию для домена
udt_catalog	sql_identifier	Имя базы данных, в которой определён тип данных домена (всегда текущая база)
udt_schema	sql_identifier	Имя схемы, в которой определён тип данных домена
udt_name	sql_identifier	Имя типа данных домена
scope_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
scope_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
scope_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
maximum_cardinality	cardinal_number	Всегда NULL, так как массивы имеют неограниченную максимальную ёмкость в Postgres Pro

Имя	Тип данных	Описание
dtd_identifier	sql_identifier	Идентификатор дескриптора типа данных целевого домена, уникальный среди всех дескрипторов типов, относящихся к домену (что тривиально, так как домен содержит только один дескриптор типа). Он в основном полезен для соединения с другими экземплярами таких идентификаторов. (Конкретный формат идентификатора не определён и не гарантируется, что он останется неизменным в будущих версиях.)

34.23. element_types

Представление element_types показывает дескрипторы типов элементов массива. Когда столбец таблицы, атрибут составного типа, параметр или результат функции объявлены с типом массива, соответствующее представление информационной схемы будет содержать только ARRAY в столбце data_type. Чтобы получить информацию о типе элемента массива, можно связать соответствующее представление с данным. Например, просмотреть столбцы таблицы с типами данных и типами элементов массива (если это применимо) можно так:

```
SELECT c.column_name, c.data_type, e.data_type AS element_type
FROM information_schema.columns c LEFT JOIN information_schema.element_types e
     ON ((c.table_catalog, c.table_schema, c.table_name, 'TABLE', c.dtd_identifier)
       = (e.object_catalog, e.object_schema, e.object_name, e.object_type,
 e.collection_type_identifier))
WHERE c.table_schema = '...' AND c.table_name = '...'
ORDER BY c.ordinal_position;
```

Это представление показывает только те объекты, к которым имеет доступ текущий пользователь, являясь владельцем или имея некоторые права.

Таблица 34.21. Столбцы element_types

Имя	Тип данных	Описание
object_catalog	sql_identifier	Имя базы данных, содержащей объект, связанный с описываемым массивом (всегда текущая база)
object_schema	sql_identifier	Имя схемы, содержащей объект, связанный с описываемым массивом
object_name	sql_identifier	Имя объекта, связанного с описываемым массивом
object_type	character_data	Тип объекта, связанного с описываемым массивом: TABLE (массив задействован в столбце этой таблицы), USER-DEFINED TYPE (массив задействован в атрибуте составного типа), DOMAIN (массив задействован в домене), ROUTINE (массив

Имя	Тип данных	Описание
		задействован в типе данных параметра или результата функции).
collection_type_identifier	sql_identifier	Идентификатор дескриптора типа данных для описываемого массива. Его можно использовать для соединения со столбцами dtd_identifier других представлений информационной схемы.
data_type	character_data	Тип данных элементов массива, если это встроенный тип, либо USER-DEFINED (в этом случае тип определяется в udt_name и связанных столбцах).
character_maximum_length	cardinal_number	Всегда NULL, так как эта информация неприменима к типам элементов массива в Postgres Pro
character_octet_length	cardinal_number	Всегда NULL, так как эта информация неприменима к типам элементов массива в Postgres Pro
character_set_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
character_set_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
character_set_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
collation_catalog	sql_identifier	Имя базы данных, содержащей правило сортировки типа элемента (это всегда текущая база), либо NULL, если это правило по умолчанию или тип элемента несортируемый
collation_schema	sql_identifier	Имя схемы, содержащей правило сортировки типа элемента, либо NULL, если это правило по умолчанию или тип элемента несортируемый
collation_name	sql_identifier	Имя правила сортировки типа элемента, либо NULL, если это правило по умолчанию или тип элемента несортируемый
numeric_precision	cardinal_number	Всегда NULL, так как эта информация неприменима к типам элементов массива в Postgres Pro

Имя	Тип данных	Описание
numeric_precision_radix	cardinal_number	Всегда NULL, так как эта информация неприменима к типам элементов массива в Postgres Pro
numeric_scale	cardinal_number	Всегда NULL, так как эта информация неприменима к типам элементов массива в Postgres Pro
datetime_precision	cardinal_number	Всегда NULL, так как эта информация неприменима к типам элементов массива в Postgres Pro
interval_type	character_data	Всегда NULL, так как эта информация неприменима к типам элементов массива в Postgres Pro
interval_precision	cardinal_number	Всегда NULL, так как эта информация неприменима к типам элементов массива в Postgres Pro
domain_default	character_data	Ещё не реализовано
udt_catalog	sql_identifier	Имя базы данных, в которой определён тип данных элемента (всегда текущая база)
udt_schema	sql_identifier	Имя схемы, в которой определён тип данных элемента
udt_name	sql_identifier	Имя типа данных элемента
scope_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
scope_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
scope_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
maximum_cardinality	cardinal_number	Всегда NULL, так как массивы имеют неограниченную максимальную ёмкость в Postgres Pro
dtd_identifier	sql_identifier	Идентификатор дескриптора типа данных элемента. В настоящее время бесполезен.

34.24. enabled_roles

Представление enabled_roles описывает «доступные роли». Список доступных ролей рекурсивно определяется как роль текущего пользователя плюс роли, данные доступным ролям, с автоматическим наследованием. Другими словами, это роли, которые даны текущему пользователю непосредственно или косвенно, посредством автоматического наследования.

Для проверки разрешений применяется набор «применимых ролей», который может быть шире набора доступных ролей. Поэтому в общем случае вместо этого представления лучше использовать представление `applicable_roles` (оно описывается в Разделе 34.5).

Таблица 34.22. Столбцы `enabled_roles`

Имя	Тип данных	Описание
`role_name`	`sql_identifier`	Имя целевой роли

34.25. `foreign_data_wrapper_options`

Представление `foreign_data_wrapper_options` показывает все параметры, определённые для обёрток сторонних данных в текущей базе. В нём отражаются только те обёртки сторонних данных, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица 34.23. Столбцы `foreign_data_wrapper_options`

Имя	Тип данных	Описание
`foreign_data_wrapper_catalog`	`sql_identifier`	Имя базы данных, в которой определена обёртка сторонних данных (всегда текущая база)
`foreign_data_wrapper_name`	`sql_identifier`	Имя обёртки сторонних данных
`option_name`	`sql_identifier`	Имя параметра
`option_value`	`character_data`	Значение параметра

34.26. `foreign_data_wrappers`

Представление `foreign_data_wrappers` показывает все обёртки сторонних данных, определённые в текущей базе. В нём показываются только те обёртки сторонних данных, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица 34.24. Столбцы `foreign_data_wrappers`

Имя	Тип данных	Описание
`foreign_data_wrapper_catalog`	`sql_identifier`	Имя базы данных, в которой определена обёртка сторонних данных (всегда текущая база)
`foreign_data_wrapper_name`	`sql_identifier`	Имя обёртки сторонних данных
`authorization_identifier`	`sql_identifier`	Имя владельца стороннего сервера
`library_name`	`character_data`	Имя файла библиотеки, реализующей эту обёртку сторонних данных
`foreign_data_wrapper_language`	`character_data`	Язык, на котором реализована эта обёртка сторонних данных

34.27. `foreign_server_options`

Представление `foreign_server_options` показывает все параметры, определённые для сторонних серверов в текущей базе данных. В нём отражаются только те сторонние серверы, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица 34.25. Столбцы `foreign_server_options`

Имя	Тип данных	Описание
`foreign_server_catalog`	`sql_identifier`	Имя базы данных, в которой определён сторонний сервер (всегда текущая база)
`foreign_server_name`	`sql_identifier`	Имя стороннего сервера
`option_name`	`sql_identifier`	Имя параметра
`option_value`	`character_data`	Значение параметра

34.28. `foreign_servers`

Представление `foreign_servers` показывает все сторонние серверы, определённые в текущей базе данных. В нём показываются только те сторонние серверы, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица 34.26. Столбцы `foreign_servers`

Имя	Тип данных	Описание
`foreign_server_catalog`	`sql_identifier`	Имя базы данных, в которой определён сторонний сервер (всегда текущая база)
`foreign_server_name`	`sql_identifier`	Имя стороннего сервера
`foreign_data_wrapper_ catalog`	`sql_identifier`	Имя базы данных, в которой определена обёртка сторонних данных, используемая сторонним сервером (всегда текущая база)
`foreign_data_wrapper_name`	`sql_identifier`	Имя обёртки сторонних данных, используемой сторонним сервером
`foreign_server_type`	`character_data`	Информация о типе стороннего сервера, если она была указана при его создании
`foreign_server_version`	`character_data`	Информация о версии стороннего сервера, если она была указана при его создании
`authorization_identifier`	`sql_identifier`	Имя владельца стороннего сервера

34.29. `foreign_table_options`

Представление `foreign_table_options` показывает все параметры, определённые для сторонних таблиц в текущей базе данных. В нём отражаются только те сторонние таблицы, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица 34.27. Столбцы `foreign_table_options`

Имя	Тип данных	Описание
`foreign_table_catalog`	`sql_identifier`	Имя базы данных, содержащей стороннюю таблицу (всегда текущая база)
`foreign_table_schema`	`sql_identifier`	Имя схемы, содержащей стороннюю таблицу

Имя	Тип данных	Описание
foreign_table_name	sql_identifier	Имя сторонней таблицы
option_name	sql_identifier	Имя параметра
option_value	character_data	Значение параметра

34.30. foreign_tables

Представление foreign_tables показывает все сторонние таблицы, определённые в текущей базе данных. В нём показываются только те сторонние таблицы, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица 34.28. Столбцы foreign_tables

Имя	Тип данных	Описание
foreign_table_catalog	sql_identifier	Имя базы данных, в которой определена сторонняя таблица (всегда текущая база)
foreign_table_schema	sql_identifier	Имя схемы, содержащей стороннюю таблицу
foreign_table_name	sql_identifier	Имя сторонней таблицы
foreign_server_catalog	sql_identifier	Имя базы данных, в которой определён сторонний сервер (всегда текущая база)
foreign_server_name	sql_identifier	Имя стороннего сервера

34.31. key_column_usage

Представление key_column_usage описывает все столбцы в текущей базе, с которыми связано какое-либо ограничение уникальности, либо ограничение первичного или внешнего ключа. Ограничения-проверки в этом представлении не показываются. В нём показываются только те столбцы, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 34.29. Столбцы key_column_usage

Имя	Тип данных	Описание
constraint_catalog	sql_identifier	Имя базы данных, содержащей ограничение (всегда текущая база)
constraint_schema	sql_identifier	Имя схемы, содержащей ограничение
constraint_name	sql_identifier	Имя ограничения
table_catalog	sql_identifier	Имя базы данных, содержащей таблицу со столбцом, подчиняющимся этому ограничению (всегда текущая база)
table_schema	sql_identifier	Имя схемы, содержащей таблицу со столбцом, подчиняющимся этому ограничению
table_name	sql_identifier	Имя таблицы со столбцом, подчиняющимся этому ограничению

Имя	Тип данных	Описание
column_name	sql_identifier	Имя столбца, подчиняющегося этому ограничению
ordinal_position	cardinal_number	Порядковый номер столбца в ключе ограничения (нумерация начинается с 1)
position_in_unique_constraint	cardinal_number	Для ограничения внешнего ключа это порядковый номер целевого столбца в его ограничении уникальности (нумерация начинается с 1); в противном случае NULL

34.32. parameters

Представление parameters содержит информацию о параметрах (аргументах) всех функций в текущей базе данных. В нём отражаются только функции, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 34.30. Столбцы parameters

Имя	Тип данных	Описание
specific_catalog	sql_identifier	Имя базы данных, содержащей функцию (всегда текущая база)
specific_schema	sql_identifier	Имя схемы, содержащей функцию
specific_name	sql_identifier	«Однозначное имя» функции. Подробнее оно описано в Разделе 34.40.
ordinal_position	cardinal_number	Порядковый номер параметра в списке аргументов функции (нумерация начинается с 1)
parameter_mode	character_data	IN для входного параметра, OUT для выходного, INOUT — для входного и выходного параметра.
is_result	yes_or_no	Относится к функциональности, отсутствующей в Postgres Pro
as_locator	yes_or_no	Относится к функциональности, отсутствующей в Postgres Pro
parameter_name	sql_identifier	Имя параметра, либо NULL, если параметр безымянный
data_type	character_data	Тип данных параметра, если это встроенный тип, либо ARRAY, если это массив (в этом случае обратитесь к представлению element_types), иначе — USER-DEFINED (в этом случае тип определяется в udt_name и связанных столбцах).

Имя	Тип данных	Описание
character_maximum_length	cardinal_number	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
character_octet_length	cardinal_number	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
character_set_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
character_set_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
character_set_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
collation_catalog	sql_identifier	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
collation_schema	sql_identifier	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
collation_name	sql_identifier	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
numeric_precision	cardinal_number	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
numeric_precision_radix	cardinal_number	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
numeric_scale	cardinal_number	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
datetime_precision	cardinal_number	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
interval_type	character_data	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
interval_precision	cardinal_number	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
udt_catalog	sql_identifier	Имя базы данных, в которой определён тип данных параметра (всегда текущая база)
udt_schema	sql_identifier	Имя схемы, в которой определён тип данных параметра
udt_name	sql_identifier	Имя типа данных параметра

Имя	Тип данных	Описание
scope_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
scope_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
scope_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
maximum_cardinality	cardinal_number	Всегда NULL, так как массивы имеют неограниченную максимальную ёмкость в Postgres Pro
dtd_identifier	sql_identifier	Идентификатор дескриптора типа данных параметра, уникальный среди всех дескрипторов типов, относящихся к функции. Он в основном полезен для соединения с другими экземплярами таких идентификаторов. (Конкретный формат идентификатора не определён и не гарантируется, что он останется неизменным в будущих версиях.)
parameter_default	character_data	Выражение параметра по умолчанию, либо NULL, если такого выражения нет или функция не принадлежит текущей активной роли.

34.33. referential_constraints

Представление referential_constraints содержит все ссылочные ограничения (внешнего ключа) в текущей базе данных. В нём показываются только ограничения, в которых ссылающаяся таблица доступна текущему пользователю на запись (он является её владельцем или имеет не только право SELECT).

Таблица 34.31. Столбцы referential_constraints

Имя	Тип данных	Описание
constraint_catalog	sql_identifier	Имя базы данных, содержащей ограничение (всегда текущая база)
constraint_schema	sql_identifier	Имя схемы, содержащей ограничение
constraint_name	sql_identifier	Имя ограничения
unique_constraint_catalog	sql_identifier	Имя базы данных, содержащей ограничение уникальности или первичный ключ, на которые ссылается

Имя	Тип данных	Описание
		ограничение внешнего ключа (всегда текущая база)
unique_constraint_schema	sql_identifier	Имя схемы, содержащей ограничение уникальности или первичный ключ, на которые ссылается ограничение внешнего ключа
unique_constraint_name	sql_identifier	Имя ограничения уникальности или первичного ключа, на которые ссылается ограничение внешнего ключа
match_option	character_data	Тип совпадения для ограничения внешнего ключа: FULL, PARTIAL или NONE.
update_rule	character_data	Правило изменения для ограничения внешнего ключа: CASCADE, SET NULL, SET DEFAULT, RESTRICT или NO ACTION.
delete_rule	character_data	Правило удаления для ограничения внешнего ключа: CASCADE, SET NULL, SET DEFAULT, RESTRICT или NO ACTION.

34.34. role_column_grants

Представление role_column_grants описывает все назначенные для столбцов права, в которых праводателем или правообладателем является текущая активная роль. Дополнительную информацию можно найти в column_privileges. Единственное существенное отличие этого представления от column_privileges состоит в том, что в данном представлении опускаются столбцы, которые доступны текущему пользователю косвенно через роль PUBLIC.

Таблица 34.32. Столбцы role_column_grants

Имя	Тип данных	Описание
grantor	sql_identifier	Имя роли, давшей право (праводатель)
grantee	sql_identifier	Имя роли, которой было дано право (правообладатель)
table_catalog	sql_identifier	Имя базы данных, содержащей таблицу с этим столбцом (всегда текущая база)
table_schema	sql_identifier	Имя схемы, содержащей таблицу с этим столбцом
table_name	sql_identifier	Имя таблицы с этим столбцом
column_name	sql_identifier	Имя столбца
privilege_type	character_data	Тип права: SELECT, INSERT, UPDATE или REFERENCES
is_grantable	yes_or_no	YES, если право может передаваться, или NO в противном случае

34.35. role_routine_grants

Представление role_routine_grants описывает все назначенные для функций права, в которых праводателем или правообладателем является текущая активная роль. Дополнительную информацию можно найти в routine_privileges. Единственное существенное отличие этого представления от routine_privileges состоит в том, что в данном представлении опускаются функции, которые доступны текущему пользователю косвенно через роль PUBLIC.

Таблица 34.33. Столбцы role_routine_grants

Имя	Тип данных	Описание
grantor	sql_identifier	Имя роли, давшей право (праводатель)
grantee	sql_identifier	Имя роли, которой было дано право (правообладатель)
specific_catalog	sql_identifier	Имя базы данных, содержащей функцию (всегда текущая база)
specific_schema	sql_identifier	Имя схемы, содержащей функцию
specific_name	sql_identifier	«Однозначное имя» функции. Подробнее оно описано в Разделе 34.40.
routine_catalog	sql_identifier	Имя базы данных, содержащей функцию (всегда текущая база)
routine_schema	sql_identifier	Имя схемы, содержащей функцию
routine_name	sql_identifier	Имя функции (может дублироваться в случае перегрузки)
privilege_type	character_data	Всегда EXECUTE (единственный тип прав для функций)
is_grantable	yes_or_no	YES, если право может передаваться, или NO в противном случае

34.36. role_table_grants

Представление role_table_grants описывает все назначенные для таблиц и представлений права, в которых праводателем или правообладателем является текущая активная роль. Дополнительную информацию можно найти в table_privileges. Единственное существенное отличие этого представления от table_privileges состоит в том, что в данном представлении опускаются таблицы, которые доступны текущему пользователю косвенно через роль PUBLIC.

Таблица 34.34. Столбцы role_table_grants

Имя	Тип данных	Описание
grantor	sql_identifier	Имя роли, давшей право (праводатель)
grantee	sql_identifier	Имя роли, которой было дано право (правообладатель)
table_catalog	sql_identifier	Имя базы данных, содержащей таблицу (всегда текущая база)
table_schema	sql_identifier	Имя схемы, содержащей таблицу

Имя	Тип данных	Описание
table_name	sql_identifier	Имя таблицы
privilege_type	character_data	Тип права: SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES или TRIGGER
is_grantable	yes_or_no	YES, если право может передаваться, или NO в противном случае
with_hierarchy	yes_or_no	В стандарте SQL имеется отдельное подчинённое разрешение WITH HIERARCHY OPTION, позволяющее выполнять определённые операции в иерархии наследования таблиц. В Postgres Pro так действует право SELECT, так что в этом столбце выводится YES для права SELECT, а для других — NO.

34.37. role_udt_grants

Представление role_udt_grants предназначено для отображения прав USAGE, назначенных для пользовательских типов, в которых праводателем или правообладателем является текущая активная роль. Дополнительную информацию можно найти в udt_privileges. Единственное существенное отличие этого представления от udt_privileges состоит в том, что в данном представлении опускаются объекты, которые доступны текущему пользователю косвенно через роль PUBLIC. Так как с типами данных не связываются действующие права в Postgres Pro (только PUBLIC неявно даётся право их использования), это представление пустое.

Таблица 34.35. Столбцы role_udt_grants

Имя	Тип данных	Описание
grantor	sql_identifier	Имя роли, которая дала это право
grantee	sql_identifier	Имя роли, которой было дано это право
udt_catalog	sql_identifier	Имя базы данных, содержащей тип (всегда текущая база)
udt_schema	sql_identifier	Имя схемы, содержащей тип
udt_name	sql_identifier	Имя типа
privilege_type	character_data	Всегда TYPE USAGE
is_grantable	yes_or_no	YES, если право может передаваться, или NO в противном случае

34.38. role_usage_grants

Представление role_usage_grants описывает права USAGE, назначенные для объектов различных типов, в которых праводателем или правообладателем является текущая активная роль. Дополнительную информацию можно найти в usage_privileges. Единственное существенное отличие этого представления от usage_privileges состоит в том, что в данном представлении опускаются объекты, которые доступны текущему пользователю косвенно через роль PUBLIC.

Таблица 34.36. Столбцы `role_usage_grants`

Имя	Тип данных	Описание
`grantor`	`sql_identifier`	Имя роли, которая дала это право
`grantee`	`sql_identifier`	Имя роли, которой было дано это право
`object_catalog`	`sql_identifier`	Имя базы данных, содержащей объект (всегда текущая база)
`object_schema`	`sql_identifier`	Имя схемы, содержащей объект, если это применимо, иначе пустая строка
`object_name`	`sql_identifier`	Имя объекта
`object_type`	`character_data`	`COLLATION` или `DOMAIN` или `FOREIGN DATA WRAPPER` или `FOREIGN SERVER` или `SEQUENCE`
`privilege_type`	`character_data`	Всегда `USAGE`
`is_grantable`	`yes_or_no`	`YES`, если право может передаваться, или `NO` в противном случае

34.39. `routine_privileges`

Представление `routine_privileges` описывает все права, назначенные текущей активной роли или текущей активной ролью для функций. Оно содержит отдельный столбец для каждой комбинации функции, праводателя и правообладателя.

Таблица 34.37. Столбцы `routine_privileges`

Имя	Тип данных	Описание
`grantor`	`sql_identifier`	Имя роли, давшей право (праводатель)
`grantee`	`sql_identifier`	Имя роли, которой было дано право (правообладатель)
`specific_catalog`	`sql_identifier`	Имя базы данных, содержащей функцию (всегда текущая база)
`specific_schema`	`sql_identifier`	Имя схемы, содержащей функцию
`specific_name`	`sql_identifier`	«Однозначное имя» функции. Подробнее оно описано в Разделе 34.40.
`routine_catalog`	`sql_identifier`	Имя базы данных, содержащей функцию (всегда текущая база)
`routine_schema`	`sql_identifier`	Имя схемы, содержащей функцию
`routine_name`	`sql_identifier`	Имя функции (может дублироваться в случае перегрузки)
`privilege_type`	`character_data`	Всегда `EXECUTE` (единственный тип прав для функций)

Имя	Тип данных	Описание
is_grantable	yes_or_no	YES, если право может передаваться, или NO в противном случае

34.40. routines

Представление routines показывает все функции в текущей базе данных. В нём показываются только функции, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 34.38. Столбцы routines

Имя	Тип данных	Описание
specific_catalog	sql_identifier	Имя базы данных, содержащей функцию (всегда текущая база)
specific_schema	sql_identifier	Имя схемы, содержащей функцию
specific_name	sql_identifier	«Однозначное имя» функции. Это имя однозначным образом идентифицирует функцию в схеме, даже если реальное имя функции перегружено. Формат однозначных имён не определён, так что его следует использовать только для сравнения с другими экземплярами однозначных имён подпрограмм.
routine_catalog	sql_identifier	Имя базы данных, содержащей функцию (всегда текущая база)
routine_schema	sql_identifier	Имя схемы, содержащей функцию
routine_name	sql_identifier	Имя функции (может дублироваться в случае перегрузки)
routine_type	character_data	Всегда FUNCTION (в будущем могут появиться и другие типы подпрограмм.)
module_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
module_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
module_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
udt_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro

Имя	Тип данных	Описание
udt_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
udt_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
data_type	character_data	Тип данных результата функции, если это встроенный тип, либо ARRAY, если это массив (в этом случае обратитесь к представлению element_types), иначе — USER-DEFINED (в этом случае тип определяется в type_udt_name и связанных столбцах).
character_maximum_length	cardinal_number	Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
character_octet_length	cardinal_number	Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
character_set_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
character_set_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
character_set_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
collation_catalog	sql_identifier	Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
collation_schema	sql_identifier	Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
collation_name	sql_identifier	Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
numeric_precision	cardinal_number	Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
numeric_precision_radix	cardinal_number	Всегда NULL, так как эта информация неприменима к

Имя	Тип данных	Описание
		типам результатов в Postgres Pro
numeric_scale	cardinal_number	Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
datetime_precision	cardinal_number	Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
interval_type	character_data	Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
interval_precision	cardinal_number	Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
type_udt_catalog	sql_identifier	Имя базы данных, в которой определён тип данных результата функции (всегда текущая база)
type_udt_schema	sql_identifier	Имя схемы, в которой определён тип данных результата функции
type_udt_name	sql_identifier	Имя типа данных результата функции
scope_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
scope_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
scope_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
maximum_cardinality	cardinal_number	Всегда NULL, так как массивы имеют неограниченную максимальную ёмкость в Postgres Pro
dtd_identifier	sql_identifier	Идентификатор дескриптора типа данных результата функции, уникальный среди всех дескрипторов типов, относящихся к функции. Он в основном полезен для соединения с другими экземплярами таких идентификаторов. (Конкретный формат идентификатора не определён и не гарантируется,

Имя	Тип данных	Описание
		что он останется неизменным в будущих версиях.)
routine_body	character_data	Если функция написана на SQL, это поле содержит SQL, иначе EXTERNAL.
routine_definition	character_data	Исходный текст функции (NULL, если функция не принадлежит текущей активной роли). (Согласно стандарту SQL, этот столбец актуален, только если в routine_body указано SQL, но в Postgres Pro он будет содержать любой исходный текст, заданный при создании функции.)
external_name	character_data	Если это функция на C, этот столбец содержит внешнее имя (объектный символ) функции, иначе — NULL. (Это будет то же значение, что содержит столбец routine_definition .)
external_language	character_data	Язык, на котором написана функция
parameter_style	character_data	Всегда GENERAL (В стандарте SQL определены и другие стили параметров, но в Postgres Pro они отсутствуют.)
is_deterministic	yes_or_no	Если функция объявлена как постоянная (IMMUTABLE) (в стандарте SQL она называется детерминированной), этот столбец содержит YES, иначе — NO. (Получить другие уровни переменности функций, имеющиеся в Postgres Pro, через информационную схему нельзя.)
sql_data_access	character_data	Всегда MODIFIES, что означает, что функция может модифицировать данные SQL. Для Postgres Pro эта информация бесполезна.
is_null_call	yes_or_no	Если функция автоматически возвращает NULL, когда один из аргументов NULL, этот столбец содержит YES, иначе — NO.
sql_path	character_data	Относится к функциональности, отсутствующей в Postgres Pro

Имя	Тип данных	Описание
schema_level_routine	yes_or_no	Всегда YES (Другое значение было бы у методов пользовательских типов, но в Postgres Pro их нет.)
max_dynamic_result_sets	cardinal_number	Относится к функциональности, отсутствующей в Postgres Pro
is_user_defined_cast	yes_or_no	Относится к функциональности, отсутствующей в Postgres Pro
is_implicitly_invocable	yes_or_no	Относится к функциональности, отсутствующей в Postgres Pro
security_type	character_data	Если функция выполняется с правами вызывающего пользователя, этот столбец содержит INVOKER, а если с правами пользователя, создавшего её, то — DEFINER.
to_sql_specific_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
to_sql_specific_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
to_sql_specific_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
as_locator	yes_or_no	Относится к функциональности, отсутствующей в Postgres Pro
created	time_stamp	Относится к функциональности, отсутствующей в Postgres Pro
last_altered	time_stamp	Относится к функциональности, отсутствующей в Postgres Pro
new_savepoint_level	yes_or_no	Относится к функциональности, отсутствующей в Postgres Pro
is_udt_dependent	yes_or_no	В настоящее время всегда NO. Альтернативное значение YES связано с возможностями, отсутствующими в Postgres Pro.
result_cast_from_data_type	character_data	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_as_locator	yes_or_no	Относится к функциональности, отсутствующей в Postgres Pro

Имя	Тип данных	Описание
result_cast_char_max_length	cardinal_number	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_char_octet_length	character_data	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_char_set_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_char_set_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_char_set_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_collation_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_collation_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_collation_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_numeric_precision	cardinal_number	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_numeric_precision_radix	cardinal_number	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_numeric_scale	cardinal_number	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_datetime_precision	character_data	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_interval_type	character_data	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_interval_precision	cardinal_number	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_type_udt_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_type_udt_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro

Имя	Тип данных	Описание
result_cast_type_udt_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_scope_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_scope_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_scope_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_maximum_cardinality	cardinal_number	Относится к функциональности, отсутствующей в Postgres Pro
result_cast_dtd_identifier	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro

34.41. schemata

Представление schemata показывает все схемы в текущей базе данных, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица 34.39. Столбцы schemata

Имя	Тип данных	Описание
catalog_name	sql_identifier	Имя базы данных, содержащей схему (всегда текущая база)
schema_name	sql_identifier	Имя схемы
schema_owner	sql_identifier	Имя владельца схемы
default_character_set_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
default_character_set_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
default_character_set_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
sql_path	character_data	Относится к функциональности, отсутствующей в Postgres Pro

34.42. sequences

Представление sequences показывает все последовательности, определённые в текущей базе данных. В нём показываются только те последовательности, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 34.40. Столбцы `sequences`

Имя	Тип данных	Описание
`sequence_catalog`	`sql_identifier`	Имя базы данных, содержащей последовательность (всегда текущая база)
`sequence_schema`	`sql_identifier`	Имя схемы, содержащей последовательность
`sequence_name`	`sql_identifier`	Имя последовательности
`data_type`	`character_data`	Тип данных последовательности.
`numeric_precision`	`cardinal_number`	Этот столбец содержит точность (объявленную или неявную) типа данных последовательности (см. выше). Точность определяет число значащих цифр. Она может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу `numeric_precision_radix`.
`numeric_precision_radix`	`cardinal_number`	Этот столбец определяет, по какому основанию задаются значения в столбцах `numeric_precision` и `numeric_scale`. Возможные варианты: 2 и 10.
`numeric_scale`	`cardinal_number`	Этот столбец содержит масштаб (объявленный или неявный) типа данных последовательности (см. выше). Масштаб определяет число значащих цифр справа от десятичной точки. Он может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу `numeric_precision_radix`.
`start_value`	`character_data`	Начальное значение последовательности
`minimum_value`	`character_data`	Минимальное значение последовательности
`maximum_value`	`character_data`	Максимальное значение последовательности
`increment`	`character_data`	Шаг увеличения последовательности
`cycle_option`	`yes_or_no`	`YES`, если последовательность зацикливается, или `NO` в противном случае

Заметьте, что, согласно стандарту SQL, начальное, минимальное, максимальное значение и шаг выдаются в виде символьных строк.

34.43. sql_features

Таблица sql_features содержит информацию о формальной функциональности, описанной в стандарте и поддерживаемой Postgres Pro. Эта же информация представлена в Приложении D. Там вы можете найти и дополнительные сведения.

Таблица 34.41. Столбцы sql_features

Имя	Тип данных	Описание
feature_id	character_data	Строка идентификатора функциональности
feature_name	character_data	Описательное название функциональности
sub_feature_id	character_data	Строка идентификатора подчинённой возможности, либо строка нулевой длины, если это не подчинённая возможность
sub_feature_name	character_data	Описательное название подчинённой возможности, либо строка нулевой длины, если это не подчинённая возможность
is_supported	yes_or_no	YES, если функциональность полностью поддерживается текущей версией Postgres Pro, либо NO в противном случае
is_verified_by	character_data	Всегда NULL, так как группа разработки Postgres Pro не проводит формальное тестирование функционального соответствия
comments	character_data	Необязательный комментарий о поддерживаемом состоянии функциональности

34.44. sql_implementation_info

Таблица sql_implementation_info содержит информацию о различных аспектах, которые в стандарте SQL оставлены на усмотрение реализации. Эта информация в основном предназначена для применения в контексте интерфейса ODBC; пользователям других интерфейсов она вряд ли будет нужна. Поэтому отдельные элементы особенностей реализации здесь не описываются; о них можно прочитать в описании интерфейса ODBC.

Таблица 34.42. Столбцы sql_implementation_info

Имя	Тип данных	Описание
implementation_info_id	character_data	Строка идентификатора элемента особенности реализации
implementation_info_name	character_data	Описательное название элемента особенности реализации
integer_value	cardinal_number	Значение элемента особенности реализации, либо

Имя	Тип данных	Описание
		NULL, если его значение содержится в столбце character_value
character_value	character_data	Значение элемента особенности реализации, либо NULL, если его значение содержится в столбце integer_value
comments	character_data	Необязательный комментарий, относящийся к элементу особенности реализации

34.45. sql_languages

Таблица sql_languages содержит одну строку для каждой привязки языка SQL, поддерживаемой Postgres Pro. Postgres Pro поддерживает непосредственный SQL и SQL, встроенный в C; это всё, что можно получить из этой таблицы.

Эта таблица была исключена из стандарта SQL в версии SQL:2008, так что она не содержит записей, относящихся к стандартам новее SQL:2003.

Таблица 34.43. Столбцы sql_languages

Имя	Тип данных	Описание
sql_language_source	character_data	Имя источника определения языка; всегда ISO 9075, то есть стандарт SQL
sql_language_year	character_data	Год, в котором был одобрен стандарт, отмеченный в sql_language_source .
sql_language_conformance	character_data	Уровень совместимости со стандартом для привязки языка. Для ISO 9075:2003 это всегда CORE.
sql_language_integrity	character_data	Всегда NULL (Это значение было актуально для более ранней версии стандарта SQL.)
sql_language_implementation	character_data	Всегда NULL
sql_language_binding_style	character_data	Стиль привязки языка: DIRECT или EMBEDDED
sql_language_programming_language	character_data	Язык программирования, если стиль привязки — EMBEDDED, иначе NULL. Postgres Pro поддерживает только язык C.

34.46. sql_packages

Таблица sql_packages содержит информацию о функциональных пакетах, определённых в стандарте SQL и поддерживаемых Postgres Pro. За дополнительными сведениями о функциональных пакетах обратитесь к Приложению D.

Таблица 34.44. Столбцы sql_packages

Имя	Тип данных	Описание
feature_id	character_data	Строка идентификатора пакета
feature_name	character_data	Описательное название пакета
is_supported	yes_or_no	YES, если пакет полностью поддерживается текущей версией Postgres Pro, либо NO в противном случае
is_verified_by	character_data	Всегда NULL, так как группа разработки Postgres Pro не проводит формальное тестирование функционального соответствия
comments	character_data	Необязательный комментарий о поддерживаемом состоянии пакета

34.47. sql_parts

Таблица sql_parts содержит информацию о различных частях стандарта SQL, поддерживаемых Postgres Pro.

Таблица 34.45. Столбцы sql_parts

Имя	Тип данных	Описание
feature_id	character_data	Строка идентификатора, содержащая номер части
feature_name	character_data	Описательное название части
is_supported	yes_or_no	YES, если часть полностью поддерживается текущей версией Postgres Pro, либо NO в противном случае
is_verified_by	character_data	Всегда NULL, так как группа разработки Postgres Pro не проводит формальное тестирование функционального соответствия
comments	character_data	Необязательный комментарий о поддерживаемом состоянии части

34.48. sql_sizing

Таблица sql_sizing содержит информацию о различных ограничениях размера и максимальных значениях в Postgres Pro. Эта информация в основном предназначена для применения в контексте интерфейса ODBC; пользователям других интерфейсов она вряд ли будет нужна. Поэтому отдельные элементы размеров здесь не описываются; о них можно прочитать в описании интерфейса ODBC.

Таблица 34.46. Столбцы sql_sizing

Имя	Тип данных	Описание
sizing_id	cardinal_number	Идентификатор элемента размеров

Имя	Тип данных	Описание
sizing_name	character_data	Описательное название элемента размеров
supported_value	cardinal_number	Значение элемента размеров, или 0, если размер неограниченный или не может быть определён, либо NULL, если функциональность, к которой относится размер, не поддерживается
comments	character_data	Необязательный комментарий, относящийся к элементу размеров

34.49. sql_sizing_profiles

Таблица sql_sizing_profiles содержит информацию о значениях sql_sizing, которые требуются для различных профилей стандарта SQL. Postgres Pro не ориентируется на профили SQL, поэтому данная таблица пуста.

Таблица 34.47. Столбцы sql_sizing_profiles

Имя	Тип данных	Описание
sizing_id	cardinal_number	Идентификатор элемента размеров
sizing_name	character_data	Описательное название элемента размеров
profile_id	character_data	Строка идентификатора профиля
required_value	cardinal_number	Значение, которое требует профиль SQL для этого элемента размеров, или 0, если профиль не ограничивает данный элемент, либо NULL, если профиль не требует наличия функциональности, для которой применим этот элемент
comments	character_data	Необязательный комментарий, относящийся к элементу размеров в профиле

34.50. table_constraints

Представление table_constraints показывает все ограничения, принадлежащие таблицам, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права, кроме SELECT).

Таблица 34.48. Столбцы table_constraints

Имя	Тип данных	Описание
constraint_catalog	sql_identifier	Имя базы данных, содержащей ограничение (всегда текущая база)

Имя	Тип данных	Описание
constraint_schema	sql_identifier	Имя схемы, содержащей ограничение
constraint_name	sql_identifier	Имя ограничения
table_catalog	sql_identifier	Имя базы данных, содержащей таблицу (всегда текущая база)
table_schema	sql_identifier	Имя схемы, содержащей таблицу
table_name	sql_identifier	Имя таблицы
constraint_type	character_data	Тип ограничения: CHECK, FOREIGN KEY, PRIMARY KEY или UNIQUE
is_deferrable	yes_or_no	YES, если ограничение откладываемое, или NO в противном случае
initially_deferred	yes_or_no	YES, если ограничение откладываемое и отложенное изначально, или NO в противном случае

34.51. table_privileges

Представление table_privileges описывает все права, назначенные текущей активной роли или текущей активной ролью для таблиц и представлений. Оно содержит отдельную строку для каждой комбинации таблицы, праводателя и правообладателя.

Таблица 34.49. Столбцы table_privileges

Имя	Тип данных	Описание
grantor	sql_identifier	Имя роли, давшей право (праводатель)
grantee	sql_identifier	Имя роли, которой было дано право (правообладатель)
table_catalog	sql_identifier	Имя базы данных, содержащей таблицу (всегда текущая база)
table_schema	sql_identifier	Имя схемы, содержащей таблицу
table_name	sql_identifier	Имя таблицы
privilege_type	character_data	Тип права: SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES или TRIGGER
is_grantable	yes_or_no	YES, если право может передаваться, или NO в противном случае
with_hierarchy	yes_or_no	В стандарте SQL имеется отдельное подчинённое разрешение WITH HIERARCHY OPTION, позволяющее выполнять определённые операции в иерархии наследования таблиц. В Postgres Pro так действует

Имя	Тип данных	Описание
		право SELECT, так что в этом столбце выводится YES для права SELECT, а для других — NO.

34.52. tables

Представление tables показывает все таблицы и представления, определённые в текущей базе данных. В нём показываются только те таблицы и представления, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица 34.50. Столбцы tables

Имя	Тип данных	Описание
table_catalog	sql_identifier	Имя базы данных, содержащей таблицу (всегда текущая база)
table_schema	sql_identifier	Имя схемы, содержащей таблицу
table_name	sql_identifier	Имя таблицы
table_type	character_data	Тип таблицы: BASE TABLE для постоянных базовых таблиц (таблиц обычного типа), VIEW для представлений, FOREIGN TABLE для сторонних таблиц, либо LOCAL TEMPORARY для временных таблиц
self_referencing_column_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
reference_generation	character_data	Относится к функциональности, отсутствующей в Postgres Pro
user_defined_type_catalog	sql_identifier	Если таблица является типизированной, это имя базы данных, содержащей нижележащий тип данных (всегда текущая база), иначе — NULL.
user_defined_type_schema	sql_identifier	Если таблица является типизированной, это имя схемы, содержащей нижележащий тип данных, иначе — NULL.
user_defined_type_name	sql_identifier	Если таблица является типизированной, это имя типа данных, иначе — NULL.
is_insertable_into	yes_or_no	YES, если в эту таблицу можно добавлять данные, или NO в противном случае (Базовые таблицы всегда допускают добавление данных, но представления — не обязательно.)

Имя	Тип данных	Описание
is_typed	yes_or_no	YES, если эта таблица является типизированной, иначе — NO
commit_action	character_data	Ещё не реализовано

34.53. transforms

Представление transforms содержит информацию о трансформациях, определённых в текущей базе данных. Более точно, оно содержит строку для каждой функции, задействованной в трансформации (функцию «из SQL» или «в SQL»).

Таблица 34.51. Столбцы transforms

Имя	Тип данных	Описание
udt_catalog	sql_identifier	Имя базы данных, содержащей тип, для которого предназначена трансформация (всегда текущая база)
udt_schema	sql_identifier	Имя схемы, содержащей тип, для которого предназначена трансформация
udt_name	sql_identifier	Имя типа, для которого предназначена трансформация
specific_catalog	sql_identifier	Имя базы данных, содержащей функцию (всегда текущая база)
specific_schema	sql_identifier	Имя схемы, содержащей функцию
specific_name	sql_identifier	«Однозначное имя» функции. Подробнее оно описано в Разделе 34.40.
group_name	sql_identifier	Стандарт SQL позволяет определять «группы» трансформаций и выбирать группу во время выполнения. Postgres Pro это не поддерживает. Вместо этого трансформации привязываются к языкам. В качестве компромисса со стандартом, это поле содержит язык, для которого предназначена трансформация.
transform_type	character_data	FROM SQL или TO SQL

34.54. triggered_update_columns

Для триггеров в текущей базе данных, установленных для списка столбцов (например, UPDATE OF column1, column2), представление triggered_update_columns показывает эти столбцы. Триггеры, для которых не задаётся список столбцов, в этом представлении не отражаются. В нём показываются только столбцы, доступные текущему пользователю (как владельцу или имеющему некоторые права, кроме SELECT).

Таблица 34.52. Столбцы `triggered_update_columns`

Имя	Тип данных	Описание
`trigger_catalog`	`sql_identifier`	Имя базы данных, содержащей триггер (всегда текущая база)
`trigger_schema`	`sql_identifier`	Имя схемы, содержащей триггер
`trigger_name`	`sql_identifier`	Имя триггера
`event_object_catalog`	`sql_identifier`	Имя базы данных, содержащей таблицу, для которой определён триггер (всегда текущая база)
`event_object_schema`	`sql_identifier`	Имя схемы, содержащей таблицу, для которой определён триггер
`event_object_table`	`sql_identifier`	Имя таблицы, для которой определён триггер
`event_object_column`	`sql_identifier`	Имя столбца, для которого определён триггер

34.55. `triggers`

Представление `triggers` показывает все триггеры, определённые в текущей базе данных для таблиц и представлений, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права, кроме `SELECT`).

Таблица 34.53. Столбцы `triggers`

Имя	Тип данных	Описание
`trigger_catalog`	`sql_identifier`	Имя базы данных, содержащей триггер (всегда текущая база)
`trigger_schema`	`sql_identifier`	Имя схемы, содержащей триггер
`trigger_name`	`sql_identifier`	Имя триггера
`event_manipulation`	`character_data`	Событие, вызывающие срабатывание триггера (`INSERT`, `UPDATE` или `DELETE`)
`event_object_catalog`	`sql_identifier`	Имя базы данных, содержащей таблицу, для которой определён триггер (всегда текущая база)
`event_object_schema`	`sql_identifier`	Имя схемы, содержащей таблицу, для которой определён триггер
`event_object_table`	`sql_identifier`	Имя таблицы, для которой определён триггер
`action_order`	`cardinal_number`	Ещё не реализовано
`action_condition`	`character_data`	Условие `WHEN` триггера, либо NULL, если его нет (так же NULL, если таблица не принадлежит текущей активной роли)
`action_statement`	`character_data`	Оператор, выполняемый триггером (в настоящее время

Имя	Тип данных	Описание
		всегда EXECUTE PROCEDURE *функция*(...))
action_orientation	character_data	Определяет, срабатывает ли триггер для каждой обрабатываемой строки или только для каждого оператора (ROW или STATEMENT)
action_timing	character_data	Момент срабатывания триггера (BEFORE (до), AFTER (после) или INSTEAD OF (вместо))
action_reference_old_table	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
action_reference_new_table	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
action_reference_old_row	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
action_reference_new_row	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
created	time_stamp	Относится к функциональности, отсутствующей в Postgres Pro

Триггеры в Postgres Pro несовместимы со стандартом в двух аспектах, которые влияют на их представление в информационной схеме. Во-первых, имена триггеров являются локальными для каждой таблицы в Postgres Pro, а не независимыми объектами схемы. Таким образом, в одной схеме могут быть дублирующиеся имена триггеров, если они принадлежат разным таблицам. (Значения trigger_catalog и trigger_schema на самом деле относятся к таблице, для которой определён триггер.) Во-вторых, триггеры в Postgres Pro могут срабатывать при нескольких событиях (например, ON INSERT OR UPDATE), тогда как стандарт SQL допускает только одно событие. Если триггер настроен на несколько событий, он представляется в информационной схеме в виде нескольких строк, по одной для каждого типа события. Вследствие этих двух особенностей, первичный ключ в представлении triggers на самом деле (trigger_catalog, trigger_schema, event_object_table, trigger_name, event_manipulation), а не (trigger_catalog, trigger_schema, trigger_name), как должно быть согласно стандарту SQL. Однако если определять триггеры в строгом соответствии со стандартом SQL (чтобы имена триггеров были уникальны в схеме и каждый триггер связывался только с одним событием), это расхождение никак не проявится.

Примечание

До PostgreSQL 9.1 в этом представлении столбцы action_timing, action_reference_old_table, action_reference_new_table, action_reference_old_row и action_reference_new_row назывались condition_timing, condition_reference_old_table, condition_reference_new_table, condition_reference_old_row и condition_reference_new_row, соответственно. Старые имена были продиктованы стандартом SQL:1999. Новые имена соответствуют стандарту SQL:2003 и более поздним.

34.56. udt_privileges

Представление udt_privileges описывает права USAGE, назначенные текущей активной роли или текущей активной ролью для пользовательских типов. Оно содержит отдельную строку для каждой комбинации типа, праводателя и правообладателя. Это представление показывает только составные типы (почему, говорится в Разделе 34.58); права, затрагивающие домены, описаны в Разделе 34.57.

Таблица 34.54. Столбцы udt_privileges

Имя	Тип данных	Описание
grantor	sql_identifier	Имя роли, давшей право (праводатель)
grantee	sql_identifier	Имя роли, которой было дано право (правообладатель)
udt_catalog	sql_identifier	Имя базы данных, содержащей тип (всегда текущая база)
udt_schema	sql_identifier	Имя схемы, содержащей тип
udt_name	sql_identifier	Имя типа
privilege_type	character_data	Всегда TYPE USAGE
is_grantable	yes_or_no	YES, если право может передаваться, или NO в противном случае

34.57. usage_privileges

Представление usage_privileges описывает права USAGE, назначенные текущей активной роли или текущей активной ролью для различных типов объектов. В Postgres Pro на данный момент это правила сортировки, домены, обёртки сторонних данных, сторонние серверы и последовательности. Оно содержит отдельную строку для каждой комбинации объекта, праводателя и правообладателя.

Так как с правилами сортировки в Postgres Pro не связаны никакие действующие права, в этом представлении показываются фиктивные права USAGE, якобы назначенные владельцем базы роли PUBLIC для всех правил сортировки. Для других типов объектов, однако, в нём показываются фактические права.

В Postgres Pro для последовательностей могут назначаться права SELECT и UPDATE, в дополнение к USAGE. Но это нестандартные права и поэтому в информационной схеме они не видны.

Таблица 34.55. Столбцы usage_privileges

Имя	Тип данных	Описание
grantor	sql_identifier	Имя роли, давшей право (праводатель)
grantee	sql_identifier	Имя роли, которой было дано право (правообладатель)
object_catalog	sql_identifier	Имя базы данных, содержащей объект (всегда текущая база)
object_schema	sql_identifier	Имя схемы, содержащей объект, если это применимо, иначе пустая строка
object_name	sql_identifier	Имя объекта

Имя	Тип данных	Описание
object_type	character_data	COLLATION или DOMAIN или FOREIGN DATA WRAPPER или FOREIGN SERVER или SEQUENCE
privilege_type	character_data	Всегда USAGE
is_grantable	yes_or_no	YES, если право может передаваться, или NO в противном случае

34.58. user_defined_types

Представление user_defined_types в данное время показывает все составные типы, определённые в текущей базе данных. В нём показываются только те типы, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

SQL знает два вида пользовательских типов: структурные типы (также называются составными типами в Postgres Pro) и отдельные типы (не реализованы в Postgres Pro). Для уверенности в будущем, нужно обратиться к столбцу user_defined_type_category, чтобы различить их. Другие пользовательские типы, как например, базовые типы и перечисления, относящиеся к расширениям Postgres Pro, в этом представлении не показываются. О доменах можно узнать в Разделе 34.22.

Таблица 34.56. Столбцы user_defined_types

Имя	Тип данных	Описание
user_defined_type_catalog	sql_identifier	Имя базы данных, содержащей тип (всегда текущая база)
user_defined_type_schema	sql_identifier	Имя схемы, содержащей тип
user_defined_type_name	sql_identifier	Имя типа
user_defined_type_category	character_data	На данный момент всегда STRUCTURED
is_instantiable	yes_or_no	Относится к функциональности, отсутствующей в Postgres Pro
is_final	yes_or_no	Относится к функциональности, отсутствующей в Postgres Pro
ordering_form	character_data	Относится к функциональности, отсутствующей в Postgres Pro
ordering_category	character_data	Относится к функциональности, отсутствующей в Postgres Pro
ordering_routine_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
ordering_routine_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
ordering_routine_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro

Имя	Тип данных	Описание
reference_type	character_data	Относится к функциональности, отсутствующей в Postgres Pro
data_type	character_data	Относится к функциональности, отсутствующей в Postgres Pro
character_maximum_length	cardinal_number	Относится к функциональности, отсутствующей в Postgres Pro
character_octet_length	cardinal_number	Относится к функциональности, отсутствующей в Postgres Pro
character_set_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
character_set_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
character_set_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
collation_catalog	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
collation_schema	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
collation_name	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
numeric_precision	cardinal_number	Относится к функциональности, отсутствующей в Postgres Pro
numeric_precision_radix	cardinal_number	Относится к функциональности, отсутствующей в Postgres Pro
numeric_scale	cardinal_number	Относится к функциональности, отсутствующей в Postgres Pro
datetime_precision	cardinal_number	Относится к функциональности, отсутствующей в Postgres Pro
interval_type	character_data	Относится к функциональности, отсутствующей в Postgres Pro
interval_precision	cardinal_number	Относится к функциональности, отсутствующей в Postgres Pro

Имя	Тип данных	Описание
source_dtd_identifier	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro
ref_dtd_identifier	sql_identifier	Относится к функциональности, отсутствующей в Postgres Pro

34.59. user_mapping_options

Представление user_mapping_options показывает все параметры, заданные для сопоставлений пользователей в текущей базе данных. В нём отражаются только сопоставления пользователей, установленные для сторонних серверов, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 34.57. Столбцы user_mapping_options

Имя	Тип данных	Описание
authorization_identifier	sql_identifier	Имя сопоставляемого пользователя, либо PUBLIC, если это сопоставление для всех
foreign_server_catalog	sql_identifier	Имя базы данных, в которой определён сторонний сервер, задействованный в сопоставлении (всегда текущая база)
foreign_server_name	sql_identifier	Имя стороннего сервера, задействованного в сопоставлении
option_name	sql_identifier	Имя параметра
option_value	character_data	Значение параметра. Этот столбец будет содержать не NULL, только если описывается сопоставление текущего пользователя, либо это сопоставление для PUBLIC, а текущий пользователь — владелец стороннего сервера или суперпользователь. Данное ограничение введено для защиты информации о пароле, сохранённой в параметрах сопоставления.

34.60. user_mappings

Представление user_mappings показывает все сопоставления пользователей, определённые в текущей базе данных. В нём показываются только сопоставления, установленные для сторонних серверов, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 34.58. Столбцы `user_mappings`

Имя	Тип данных	Описание
`authorization_identifier`	`sql_identifier`	Имя сопоставляемого пользователя, либо `PUBLIC`, если это сопоставление для всех
`foreign_server_catalog`	`sql_identifier`	Имя базы данных, в которой определён сторонний сервер, задействованный в сопоставлении (всегда текущая база)
`foreign_server_name`	`sql_identifier`	Имя стороннего сервера, задействованного в сопоставлении

34.61. `view_column_usage`

Представление `view_column_usage` описывает все столбцы, задействованные в выражении запроса представления (операторе `SELECT`, определяющем представление). Столбец выводится в этом списке, только если содержащая его таблица принадлежит текущей активной роли.

> **Примечание**
>
> Столбцы системных таблиц в этом представлении не отображаются. Это может быть исправлено позже.

Таблица 34.59. Столбцы `view_column_usage`

Имя	Тип данных	Описание
`view_catalog`	`sql_identifier`	Имя базы данных, содержащей представление (всегда текущая база)
`view_schema`	`sql_identifier`	Имя схемы, содержащей представление
`view_name`	`sql_identifier`	Имя представления
`table_catalog`	`sql_identifier`	Имя базы данных, содержащей таблицу со столбцом, задействованным в представлении (всегда текущая база)
`table_schema`	`sql_identifier`	Имя схемы, содержащей таблицу со столбцом, задействованным в представлении
`table_name`	`sql_identifier`	Имя таблицы со столбцом, задействованным в представлении
`column_name`	`sql_identifier`	Имя столбца, задействованного в представлении

34.62. `view_routine_usage`

Представление `view_routine_usage` описывает все подпрограммы (функции и процедуры), используемые в выражении запроса представления (операторе `SELECT`, определяющем

представление). Подпрограмма выводится в этом списке, только если она принадлежит текущей активной роли.

Таблица 34.60. Столбцы `view_routine_usage`

Имя	Тип данных	Описание
`table_catalog`	`sql_identifier`	Имя базы данных, содержащей представление (всегда текущая база)
`table_schema`	`sql_identifier`	Имя схемы, содержащей представление
`table_name`	`sql_identifier`	Имя представления
`specific_catalog`	`sql_identifier`	Имя базы данных, содержащей функцию (всегда текущая база)
`specific_schema`	`sql_identifier`	Имя схемы, содержащей функцию
`specific_name`	`sql_identifier`	«Однозначное имя» функции. Подробнее оно описано в Разделе 34.40.

34.63. `view_table_usage`

Представление `view_table_usage` описывает все таблицы, задействованные в выражении запроса представления (операторе `SELECT`, определяющем представление). В этом представлении показываются только таблицы, принадлежащие текущей активной роли.

> **Примечание**
>
> Системные таблицы в этом представлении не отображаются. Это может быть исправлено позже.

Таблица 34.61. Столбцы `view_table_usage`

Имя	Тип данных	Описание
`view_catalog`	`sql_identifier`	Имя базы данных, содержащей представление (всегда текущая база)
`view_schema`	`sql_identifier`	Имя схемы, содержащей представление
`view_name`	`sql_identifier`	Имя представления
`table_catalog`	`sql_identifier`	Имя базы данных, содержащей таблицу, задействованную в представлении (всегда текущая база)
`table_schema`	`sql_identifier`	Имя схемы, содержащей таблицу, задействованную в представлении
`table_name`	`sql_identifier`	Имя таблицы, задействованной в представлении

34.64. views

Представление views показывает все представления, определённые в текущей базе данных. В нём показываются только представления, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 34.62. Столбцы views

Имя	Тип данных	Описание
table_catalog	sql_identifier	Имя базы данных, содержащей представление (всегда текущая база)
table_schema	sql_identifier	Имя схемы, содержащей представление
table_name	sql_identifier	Имя представления
view_definition	character_data	Выражение запроса, определяющее представление (NULL, если представление не принадлежит текущей активной роли)
check_option	character_data	CASCADED или LOCAL, если для представления определена характеристика CHECK OPTION, и NONE в противном случае
is_updatable	yes_or_no	YES, если представление допускает изменение (команды UPDATE и DELETE), или NO в противном случае
is_insertable_into	yes_or_no	YES, если представление допускает добавление данных (команду INSERT), или NO в противном случае
is_trigger_updatable	yes_or_no	YES, если для представления определён триггер INSTEAD OF UPDATE, или NO в противном случае
is_trigger_deletable	yes_or_no	YES, если для представления определён триггер INSTEAD OF DELETE, или NO в противном случае
is_trigger_insertable_into	yes_or_no	YES, если для представления определён триггер INSTEAD OF INSERT, или NO в противном случае

Часть V. Серверное программирование

Эта часть документации посвящена расширению функциональности сервера путём реализации собственных функций, типов данных, триггеров и т. д. Это довольно сложные темы, для освоения которых рекомендуется предварительно изучить и понять всю остальную документацию для пользователей Postgres Pro. В последних главах этой части описываются языки программирования на стороне сервера, поддерживаемые дистрибутивом Postgres Pro, и рассматриваются общие вопросы, связанные с программированием на стороне сервера. Но прежде чем погружаться в этот материал, важно изучить предыдущие разделы Главы 35 (где освещаются функции).

Глава 35. Расширение SQL

В следующих разделах мы обсудим, как в Postgres Pro можно расширять язык запросов SQL, добавляя собственные:

- функции (начиная с Раздела 35.3)
- агрегатные функции (начиная с Раздела 35.10)
- типы данных (начиная с Раздела 35.11)
- операторы (начиная с Раздела 35.12)
- классы операторов для индексов (начиная с Раздела 35.14)
- пакеты связанных объектов (начиная с Раздела 35.15)

35.1. Как реализована расширяемость

Postgres Pro является расширяемым благодаря тому, что его работа управляется каталогами. Если вы знакомы с традиционными реляционными системами баз данных, вы знаете, что они хранят информацию о базах, таблицах, столбцах и т. д., в структурах, которые обычно называются системными каталогами. (В некоторых системах они называются словарями данных.) Эти каталоги представляются пользователю в виде таблиц, подобных любым другим, но СУБД ведёт в них свои внутренние записи. Ключевое отличие Postgres Pro от обычных реляционных СУБД состоит в том, что Postgres Pro хранит в этих каталогах намного больше информации: информацию не только о таблицах и столбцах, но также о типах данных, функциях, методах доступа и т. д. Эти таблицы могут быть изменены пользователями, а так как Postgres Pro в своих действиях руководствуется этими таблицами, это означает, что пользователи могут расширять Postgres Pro. Обычные же СУБД можно расширять, только модифицируя жёстко запрограммированные процедуры в исходном коде или загружая модули, специально разработанные производителем СУБД.

Кроме того, сервер Postgres Pro может динамически загружать в свой процесс код, написанный пользователем. То есть, пользователь может подключить файл с объектным кодом (например, разделяемую библиотеку), который реализует новый тип или функцию, а Postgres Pro загрузит его по мере надобности. Код, написанный на SQL, добавляется на сервер ещё проще. Эта способность менять своё поведение «на лету» делает Postgres Pro исключительно подходящим для быстрого прототипирования новых приложений и структур хранения.

35.2. Система типов Postgres Pro

Типы данных Postgres Pro делятся на базовые, составные, доменные и псевдотипы.

35.2.1. Базовые типы

Базовые типы — это типы вроде `int4`, которые реализуются ниже уровня языка SQL (обычно на низкоуровневом языке, например, C). В общих чертах они соответствуют так называемым абстрактным типам данных. Postgres Pro может работать с такими типами только через функции, предоставленные пользователем, и понимать их поведение только в той степени, в какой его опишет пользователь. Базовые типы подразделяются на скалярные типы и массивы. Для каждого скалярного типа автоматически создаётся тип массива, который может содержать множество значений этого скалярного типа переменного размера.

35.2.2. Составные типы

Составные типы, или типы строк, образуются при создании любой таблицы. С помощью команды CREATE TYPE также можно определить «независимый» составной тип, не связанный с таблицей. Составной тип представляет собой просто список типов с определёнными именами полей. Значением составного типа является строка таблицы или запись из значений полей. Пользователь может обращаться к этим полям из запросов SQL. За дополнительными сведениями о составных типах обратитесь к Разделу 8.16.

35.2.3. Домены

Домен основывается на определённом базовом типе и во многих аспектах взаимозаменяем с ним. Однако домен может иметь дополнительные ограничения, уменьшающие множество допустимых значений относительно нижележащего базового типа.

Доменные типы можно создавать, используя SQL-команду CREATE DOMAIN. Их создание и применение в этой главе не рассматривается.

35.2.4. Псевдотипы

Для специальных целей существует также несколько «псевдотипов». Псевдотипы не могут применяться в столбцах таблицы или атрибутах составных типов, но их можно использовать в объявлениях аргументов и результатов функций. Это даёт возможность выделить в системе типов специальные классы функций. Все существующие псевдотипы перечислены в Таблице 8.25.

35.2.5. Полиморфные типы

Особый интерес представляют пять псевдотипов: `anyelement`, `anyarray`, `anynonarray`, `anyenum` и `anyrange`, которые называются *полиморфными типами*. Функция, в объявлении которой используются эти типы, называется *полиморфной*. Полиморфная функция может работать со множеством различных типов данных; конкретный тип определяется в зависимости от значения, переданного при вызове.

Полиморфные аргументы и результаты связываются друг с другом и сводятся к определённому типу данных при разборе запроса, вызывающего полиморфную функцию. В каждой позиции (в аргументах или возвращаемом значении), объявленной как `anyelement`, может передаваться любой фактический тип данных, но в каждом конкретном вызове все эти фактические типы должны быть *одинаковыми*. Аналогичным образом, в каждой позиции, объявленной как `anyarray`, может передаваться любой тип данных массива, но все фактические типы должны совпадать. Так же и во всех позициях, объявленных как `anyrange`, должен передаваться одинаковый диапазонный тип. Более того, если некоторые позиции объявлены как `anyarray`, а другие как `anyelement`, то фактическим типом в позициях `anyarray` должен быть массив, элементы которого имеют тот же тип, что и значения в позициях `anyelement`. Подобным образом, если одни позиции объявлены как `anyrange`, а другие как `anyelement` или `anyarray`, фактическим типом в позициях `anyrange` должен быть диапазон, подтип которого совпадает с типом элементов в позициях `anyelement` и с типом, передаваемым в позициях `anyarray`. Псевдотип `anynonarray` обрабатывается так же, как `anyelement`, но с дополнительным ограничением — фактический тип не должен быть типом массива. Псевдотип `anyenum` тоже обрабатывается как `anyelement`, но его фактические типы ограничиваются перечислениями.

Таким образом, когда с полиморфным типом объявлено несколько аргументов, в итоге допускаются только определённые комбинации фактических типов. Например, функция, объявленная как `equal(anyelement, anyelement)`, примет в аргументах любые два значения, но только если их типы данных совпадают.

Когда с полиморфным типом объявлено возвращаемое значение функции, так же полиморфным должен быть минимум один аргумент, и фактический тип результата при конкретном вызове определится по типу фактически переданного аргумента. Например, если бы отсутствовал механизм обращения к элементам массива, его можно было бы реализовать, создав функцию `subscript(anyarray, integer) returns anyelement`. С таким объявлением первым фактическим аргументом должен быть массив, и из него будет выведен правильный тип результата при разборе запроса. В качестве другого примера можно привести функцию `f(anyarray) returns anyenum`, которая будет принимать только массивы перечислений.

В большинстве случаев при разборе функции фактический тип данных для полиморфного результата может быть выведен из аргументов, имеющих другой полиморфный тип; например, подтип `anyarray` может выводиться из `anyelement` и наоборот. Исключение представляет полиморфный результат типа `anyrange` — для него требуется аргумент типа `anyrange`; вывести его

фактический тип из типа аргументов anyarray или anyelement нельзя. Это объясняется тем, что на одном подтипе могут базироваться несколько диапазонных типов.

Заметьте, что anynonarray и anyenum представляют не отдельные типы переменных; это те же типы, что и anyelement, но с дополнительными ограничениями. Например, объявление функции f(anyelement, anyenum) равнозначно объявлению f(anyenum, anyenum): оба фактических аргумента должны быть одинаковыми типами-перечислениями.

Функции с переменным числом аргументом (описанные в Подразделе 35.4.5) тоже могут быть полиморфными: для этого их последний параметр описывается как VARIADIC anyarray. Для целей сопоставления аргументов и определения фактического типа результата такая функция представляется так же, как если бы в ней явно объявлялось нужное число параметров anynonarray.

35.3. Пользовательские функции

В Postgres Pro представлены функции четырёх видов:

- функции на языке запросов (функции, написанные на SQL) (Раздел 35.4)
- функции на процедурных языках (функции, написанные, например, на PL/pgSQL или PL/Tcl) (Раздел 35.7)
- внутренние функции (Раздел 35.8)
- функции на языке C (Раздел 35.9)

Функции любых видов могут принимать в качестве аргументов (параметров) базовые типы, составные типы или их сочетания. Кроме того, любые функции могут возвращать значения базового или составного типа. Также можно определить функции, возвращающие наборы базовых или составных значений.

Функции многих видов могут также принимать или возвращать определённые псевдотипы (например, полиморфные типы), но доступные средства для работы с ними различаются. За подробностями обратитесь к описанию конкретного вида функций.

Проще всего определить функции на языке SQL, поэтому сначала мы рассмотрим их. Многие концепции, касающиеся функций на SQL, затем распространятся и на другие виды функций.

Изучая эту главу, будет полезно обращаться к странице справки по команде CREATE FUNCTION, чтобы лучше понимать примеры. Некоторые примеры из этой главы можно найти в файлах funcs.sql и funcs.c в каталоге src/tutorial исходного кода Postgres Pro.

35.4. Функции на языке запросов (SQL)

SQL-функции выполняют произвольный список операторов SQL и возвращают результат последнего запроса в списке. В простом случае (не с множеством) будет возвращена первая строка результата последнего запроса. (Помните, что понятие «первая строка» в наборе результатов с несколькими строками определено точно, только если присутствует ORDER BY.) Если последний запрос вообще не вернёт строки, будет возвращено значение NULL.

Кроме того, можно объявить SQL-функцию как возвращающую множество (то есть, несколько строк), указав в качестве возвращаемого типа функции SETOF *некий_тип*, либо объявив её с указанием RETURNS TABLE(*столбцы*). В этом случае будут возвращены все строки результата последнего запроса. Подробнее это описывается ниже.

Тело SQL-функции должно представлять собой список SQL-операторов, разделённых точкой с запятой. Точка с запятой после последнего оператора может отсутствовать. Если только функция не объявлена как возвращающая void, последним оператором должен быть SELECT, либо INSERT, UPDATE или DELETE с предложением RETURNING.

Любой набор команд на языке SQL можно скомпоновать вместе и обозначить как функцию. Помимо запросов SELECT, эти команды могут включать запросы, изменяющие данные (INSERT, UPDATE и DELETE), а также другие SQL-команды. (В SQL-функциях нельзя использовать команды

управления транзакциями, например COMMIT, SAVEPOINT, и некоторые вспомогательные команды, в частности VACUUM.) Однако последней командой должна быть SELECT или команда с предложением RETURNING, возвращающая результат с типом возврата функции. Если же вы хотите определить функцию SQL, выполняющую действия, но не возвращающую полезное значение, вы можете объявить её как возвращающую тип void. Например, эта функция удаляет строки с отрицательным жалованьем из таблицы emp:

```
CREATE FUNCTION clean_emp() RETURNS void AS '
    DELETE FROM emp
        WHERE salary < 0;
' LANGUAGE SQL;

SELECT clean_emp();

 clean_emp
-----------


(1 row)
```

> **Примечание**
>
> Прежде чем начинается выполнение команд, разбирается всё тело SQL-функции. Когда SQL-функция содержит команды, модифицирующие системные каталоги (например, CREATE TABLE), действие таких команд не будет проявляться на стадии анализа последующих команд этой функции. Так, например, команды CREATE TABLE foo (...); INSERT INTO foo VALUES(...); не будут работать, как ожидается, если их упаковать в одну SQL-функцию, так как foo не будет существовать к моменту разбору команды INSERT. В подобных ситуациях вместо SQL-функции рекомендуется использовать PL/pgSQL.

Синтаксис команды CREATE FUNCTION требует, чтобы тело функции было записано как строковая константа. Обычно для этого удобнее всего заключать строковую константу в доллары (см. Подраздел 4.1.2.4). Если вы решите использовать обычный синтаксис с заключением строки в апострофы, вам придётся дублировать апострофы (') и обратную косую черту (\) (предполагается синтаксис спецпоследовательностей) в теле функции (см. Подраздел 4.1.2.1).

35.4.1. Аргументы SQL-функций

К аргументам SQL-функции можно обращаться в теле функции по именам или номерам. Ниже приведены примеры обоих вариантов.

Чтобы использовать имя, объявите аргумент функции как именованный, а затем просто пишите это имя в теле функции. Если имя аргумента совпадает с именем какого-либо столбца в текущей SQL-команде внутри функции, имя столбца будет иметь приоритет. Чтобы всё же перекрыть имя столбца, дополните имя аргумента именем самой функции, то есть запишите его в виде *имя_функции*.*имя_аргумента*. (Если и это имя будет конфликтовать с полным именем столбца, снова выиграет имя столбца. Неоднозначности в этом случае вы можете избежать, выбрав другой псевдоним для таблицы в SQL-команде.)

Старый подход с нумерацией позволяет обращаться к аргументам, применяя запись $*n*: $1 обозначает первый аргумент, $2 — второй и т. д. Это будет работать и в том случае, если данному аргументу назначено имя.

Если аргумент имеет составной тип, то для обращения к его атрибутам можно использовать запись с точкой, например: *аргумент*.*поле* или $1.*поле*. И опять же, при этом может потребоваться дополнить имя аргумента именем функции, чтобы сделать имя аргумента однозначным.

Аргументы SQL-функции могут использоваться только как значения данных, но не как идентификаторы. Например, это приемлемо:

```
INSERT INTO mytable VALUES ($1);
```

а это не будет работать:

```
INSERT INTO $1 VALUES (42);
```

> **Примечание**
>
> Возможность обращаться к аргументам SQL-функций по именам появилась в PostgreSQL 9.2. В функциях, которые должны работать со старыми серверами, необходимо применять запись `$n`.

35.4.2. Функции SQL с базовыми типами

Простейшая возможная функция SQL не имеет аргументов и просто возвращает базовый тип, например `integer`:

```
CREATE FUNCTION one() RETURNS integer AS $$
    SELECT 1 AS result;
$$ LANGUAGE SQL;

-- Альтернативная запись строковой константы:
CREATE FUNCTION one() RETURNS integer AS '
    SELECT 1 AS result;
' LANGUAGE SQL;

SELECT one();

 one
-----
   1
```

Заметьте, что мы определили псевдоним столбца в теле функции для её результата (дали ему имя `result`), но этот псевдоним не виден снаружи функции. Вследствие этого, столбец результата получил имя `one`, а не `result`.

Практически так же легко определяются функции SQL, которые принимают в аргументах базовые типы:

```
CREATE FUNCTION add_em(x integer, y integer) RETURNS integer AS $$
    SELECT x + y;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

Мы также можем отказаться от имён аргументов и обращаться к ним по номерам:

```
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

Вот более полезная функция, которую можно использовать, чтобы дебетовать банковский счёт:

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno;
    SELECT 1;
$$ LANGUAGE SQL;
```

Пользователь может выполнить эту функцию, чтобы дебетовать счёт 17 на 100 долларов, так:

```
SELECT tf1(17, 100.0);
```

В этом примере мы выбрали имя `accountno` для первого аргумента, но это же имя имеет столбец в таблице `bank`. В команде `UPDATE` имя `accountno` относится к столбцу `bank.accountno`, так для обращения к аргументу нужно записать `tf1.accountno`. Конечно, мы могли бы избежать этого, выбрав другое имя для аргумента.

На практике обычно желательно получать от функции более полезный результат, чем константу 1, поэтому более реалистично такое определение:

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno;
    SELECT balance FROM bank WHERE accountno = tf1.accountno;
$$ LANGUAGE SQL;
```

Эта функция изменяет баланс и возвращает полученное значение. То же самое можно сделать в одной команде, применив `RETURNING`:

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno
    RETURNING balance;
$$ LANGUAGE SQL;
```

35.4.3. Функции SQL со сложными типами

В функциях с аргументами составных типов мы должны указывать не только, какой аргумент, но и какой атрибут (поле) этого аргумента нам нужен. Например, предположим, что `emp` — таблица, содержащая данные работников, и это же имя составного типа, представляющего каждую строку таблицы. Следующая функция `double_salary` вычисляет, каким было бы чьё-либо жалование в случае увеличения вдвое:

```
CREATE TABLE emp (
    name        text,
    salary      numeric,
    age         integer,
    cubicle     point
);

INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');

CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
    SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;

SELECT name, double_salary(emp.*) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';
```

```
 name | dream
------+-------
 Bill |  8400
```

Обратите внимание на запись `$1.salary` позволяющую выбрать одно поле из значения строки аргумента. Также заметьте, что в вызывающей команде `SELECT` указание *`имя_таблицы`*`.*` выбирает всю текущую строку таблицы как составное значение. На строку таблицы можно сослаться и просто по имени таблицы, например так:

```
SELECT name, double_salary(emp) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';
```

Однако это использование считается устаревшим, так как провоцирует путаницу. (Подробнее эти две записи составных значений строки таблицы описаны в Подразделе 8.16.5.)

Иногда бывает удобно образовать составное значение аргумента на лету. Это позволяет сделать конструкция `ROW`. Например, так можно изменить данные, передаваемые функции:

```
SELECT name, double_salary(ROW(name, salary*1.1, age, cubicle)) AS dream
    FROM emp;
```

Также возможно создать функцию, возвращающую составной тип. Например, эта функция возвращает одну строку `emp`:

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT text 'None' AS name,
        1000.0 AS salary,
        25 AS age,
        point '(2,2)' AS cubicle;
$$ LANGUAGE SQL;
```

В этом примере мы задали для каждого атрибута постоянное значение, но вместо этих констант можно подставить любые вычисления.

Учтите два важных требования относительно определения функции:

- Порядок в списке выборки внутреннего запроса должен в точности совпадать с порядком следования столбцов в таблице, связанной с составным типом. (Имена столбцов, как показывает пример выше, для системы значения не имеют.)
- Вы должны привести выражения в соответствие с определением составного типа, либо вы получите такие ошибки:

  ```
  ERROR:  function declared to return emp returns varchar instead of text at column 1
  ```

  (ОШИБКА: функция, объявленная как возвращающая emp, возвращает varchar вместо text в столбце 1)

Ту же функцию можно определить другим способом:

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
```

Здесь мы записали `SELECT`, который возвращает один столбец нужного составного типа. В данной ситуации этот вариант на самом деле не лучше, но в некоторых случаях он может быть удобной альтернативой — например, если нам нужно вычислить результат, вызывая другую функцию, которая возвращает нужное составное значение.

Мы можем вызывать эту функцию напрямую, либо указав её в выражении значения:

```
SELECT new_emp();
```

```
         new_emp
-------------------------
 (None,1000.0,25,"(2,2)")
```

либо обратившись к ней, как к табличной функции:

```
SELECT * FROM new_emp();

 name | salary | age | cubicle
------+--------+-----+---------
 None | 1000.0 |  25 | (2,2)
```

Второй способ более подробно описан в Подразделе 35.4.7.

Когда используется функция, возвращающая составной тип, может возникнуть желание получить из её результата только одно поле (атрибут). Это можно сделать, применяя такую запись:

```
SELECT (new_emp()).name;

 name
------
 None
```

Дополнительные скобки необходимы во избежание неоднозначности при разборе запроса. Если вы попытаетесь выполнить запрос без них, вы получите ошибку:

```
SELECT new_emp().name;
ERROR:  syntax error at or near "."
LINE 1: SELECT new_emp().name;
                        ^
```

(ОШИБКА: синтаксическая ошибка (примерное положение: "."))

Функциональную запись также можно использовать и для извлечения атрибутов:

```
SELECT name(new_emp());

 name
------
 None
```

Как рассказывалось в Подразделе 8.16.5, запись с указанием поля и функциональная запись являются равнозначными.

Ещё один вариант использования функции, возвращающей составной тип, заключается в передаче её результата другой функции, которая принимает этот тип строки на вход:

```
CREATE FUNCTION getname(emp) RETURNS text AS $$
    SELECT $1.name;
$$ LANGUAGE SQL;

SELECT getname(new_emp());
 getname
---------
 None
(1 row)
```

35.4.4. Функции SQL с выходными параметрами

Альтернативный способ описать результаты функции — определить её с *выходными параметрами*, как в этом примере:

```
CREATE FUNCTION add_em (IN x int, IN y int, OUT sum int)
AS 'SELECT x + y'
```

```
LANGUAGE SQL;

SELECT add_em(3,7);
 add_em
--------
     10
(1 row)
```

Это по сути не отличается от версии `add_em`, показанной в Подразделе 35.4.2. Действительная ценность выходных параметров в том, что они позволяют удобным способом определить функции, возвращающие несколько столбцов. Например:

```
CREATE FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int)
AS 'SELECT x + y, x * y'
LANGUAGE SQL;

 SELECT * FROM sum_n_product(11,42);
 sum | product
-----+---------
  53 |     462
(1 row)
```

Фактически здесь мы определили анонимный составной тип для результата функции. Показанный выше пример даёт тот же конечный результат, что и команды:

```
CREATE TYPE sum_prod AS (sum int, product int);

CREATE FUNCTION sum_n_product (int, int) RETURNS sum_prod
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

Но предыдущий вариант зачастую удобнее, так как он не требует отдельно заниматься определением составного типа. Заметьте, что имена, назначаемые выходным параметрам, не просто декоративные, а определяют имена столбцов анонимного составного типа. (Если вы опустите имя выходного параметра, система выберет имя сама.)

Заметьте, что выходные параметры не включаются в список аргументов при вызове такой функции из SQL. Это объясняется тем, что Postgres Pro определяет сигнатуру вызова функции, рассматривая только входные параметры. Это также значит, что при таких операциях, как удаление функции, в ссылках на функцию учитываются только типы входных параметров. Таким образом, удалить эту конкретную функцию можно любой из этих команд:

```
DROP FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int);
DROP FUNCTION sum_n_product (int, int);
```

Параметры функции могут быть объявлены как `IN` (по умолчанию), `OUT`, `INOUT` или `VARIADIC`. Параметр `INOUT` действует как входной (является частью списка аргументов при вызове) и как выходной (часть типа записи результата). Параметры `VARIADIC` являются входными, но обрабатывается специальным образом, как описано далее.

35.4.5. Функции SQL с переменным числом аргументов

Функции SQL могут быть объявлены как принимающие переменное число аргументов, с условием, что все «необязательные» аргументы имеют один тип данных. Необязательные аргументы будут переданы такой функции в виде массива. Для этого в объявлении функции последний параметр помечается как `VARIADIC`; при этом он должен иметь тип массива. Например:

```
CREATE FUNCTION mleast(VARIADIC arr numeric[]) RETURNS numeric AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT mleast(10, -1, 5, 4.4);
```

```
 mleast
--------
     -1
(1 row)
```

По сути, все фактические аргументы, начиная с позиции `VARIADIC`, собираются в одномерный массив, как если бы вы написали

```
SELECT mleast(ARRAY[10, -1, 5, 4.4]);    -- это не будет работать
```

На самом деле так вызвать эту функцию нельзя, или, по крайней мере, это не будет соответствовать определению функции. Параметру `VARIADIC` соответствуют одно или несколько вхождений типа его элемента, но не его собственного типа.

Но иногда бывает полезно передать функции с переменными параметрами уже подготовленный массив; особенно когда одна функция с переменными параметрами хочет передавать свой массив параметров другой. Также это более безопасный способ вызывать такую функцию, существующую в схеме, где могут создавать объекты недоверенные пользователи; см. Раздел 10.3. Это можно сделать, добавив `VARIADIC` в вызов:

```
SELECT mleast(VARIADIC ARRAY[10, -1, 5, 4.4]);
```

Это предотвращает разворачивание переменного множества параметров функции в базовый тип, что позволяет сопоставить с ним значение типа массива. `VARIADIC` можно добавить только к последнему фактическому аргументу вызова функции.

Также указание `VARIADIC` даёт единственную возможность передать пустой массив функции с переменными параметрами, например, так:

```
SELECT mleast(VARIADIC ARRAY[]::numeric[]);
```

Простой вызов `SELECT mleast()` не будет работать, так как переменным параметрам должен соответствовать минимум один фактический аргумент. (Можно определить вторую функцию с таким же именем `mleast`, но без параметров, если вы хотите выполнять такие вызовы.)

Элементы массива, создаваемые из переменных параметров, считаются не имеющими собственных имён. Это означает, что передать функции с переменными параметрами именованные аргументы нельзя (см. Раздел 4.3), если только при вызове не добавлено `VARIADIC`. Например, этот вариант будет работать:

```
SELECT mleast(VARIADIC arr => ARRAY[10, -1, 5, 4.4]);
```

А эти варианты нет:

```
SELECT mleast(arr => 10);
SELECT mleast(arr => ARRAY[10, -1, 5, 4.4]);
```

35.4.6. Функции SQL со значениями аргументов по умолчанию

Функции могут быть объявлены со значениями по умолчанию для некоторых или всех входных аргументов. Значения по умолчанию подставляются, когда функция вызывается с недостаточным количеством фактических аргументов. Так как аргументы можно опускать только с конца списка фактических аргументов, все параметры после параметра со значением по умолчанию также получат значения по умолчанию. (Хотя запись с именованными аргументами могла бы ослабить это ограничение, оно всё же остаётся в силе, чтобы позиционные ссылки на аргументы оставались действительными.) Независимо от того, используете вы эту возможность или нет, она требует осторожности при вызове функций в базах данных, где одни пользователи не доверяют другим; см. Раздел 10.3.

Например:

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL
AS $$
```

```
    SELECT $1 + $2 + $3;
$$;

SELECT foo(10, 20, 30);
 foo
-----
  60
(1 row)

SELECT foo(10, 20);
 foo
-----
  33
(1 row)

SELECT foo(10);
 foo
-----
  15
(1 row)

SELECT foo();  -- не работает из-за отсутствия значения по умолчанию для первого
 аргумента
ERROR:  function foo() does not exist
```

(ОШИБКА: функция foo() не существует) Вместо ключевого слова `DEFAULT` можно использовать знак =.

35.4.7. Функции SQL, порождающие таблицы

Все функции SQL можно использовать в предложении `FROM` запросов, но наиболее полезно это для функций, возвращающих составные типы. Если функция объявлена как возвращающая базовый тип, она возвращает таблицу с одним столбцом. Если же функция объявлена как возвращающая составной тип, она возвращает таблицу со столбцами для каждого атрибута составного типа.

Например:

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT *, upper(fooname) FROM getfoo(1) AS t1;

 fooid | foosubid | fooname | upper
-------+----------+---------+-------
     1 |        1 | Joe     | JOE
(1 row)
```

Как показывает этот пример, мы можем работать со столбцами результата функции так же, как если бы это были столбцы обычной таблицы.

Заметьте, что мы получаем из данной функции только одну строку. Это объясняется тем, что мы не использовали указание `SETOF`. Оно описывается в следующем разделе.

35.4.8. Функции SQL, возвращающие множества

Когда SQL-функция объявляется как возвращающая `SETOF` *`некий_тип`*, конечный запрос функции выполняется до завершения и каждая строка выводится как элемент результирующего множества.

Это обычно используется, когда функция вызывается в предложении `FROM`. В этом случае каждая строка, возвращаемая функцией, становится строкой таблицы, появляющейся в запросе. Например, в предположении, что таблица `foo` имеет то же содержимое, что и раньше, мы выполняем:

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;
```

Тогда в ответ мы получим:

```
 fooid | foosubid | fooname
-------+----------+---------
     1 |        1 | Joe
     1 |        2 | Ed
(2 rows)
```

Также возможно выдать несколько строк со столбцами, определяемыми выходными параметрами, следующим образом:

```
CREATE TABLE tab (y int, z int);
INSERT INTO tab VALUES (1, 2), (3, 4), (5, 6), (7, 8);

CREATE FUNCTION sum_n_product_with_tab (x int, OUT sum int, OUT product int)
RETURNS SETOF record
AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;

SELECT * FROM sum_n_product_with_tab(10);
 sum | product
-----+---------
  11 |      10
  13 |      30
  15 |      50
  17 |      70
(4 rows)
```

Здесь ключевая особенность заключается в записи `RETURNS SETOF record`, показывающей, что функция возвращает множество строк вместо одной. Если существует только один выходной параметр, укажите тип этого параметра вместо `record`.

Часто бывает полезно сконструировать результат запроса, вызывая функцию, возвращающую множество, несколько раз, передавая при каждом вызове параметры из очередных строк таблицы или подзапроса. Для этого рекомендуется применить ключевое слово `LATERAL`, описываемое в Подразделе 7.2.1.5. Ниже приведён пример использования функции, возвращающей множество, для перечисления элементов древовидной структуры:

```
SELECT * FROM nodes;
   name    | parent
-----------+--------
 Top       |
 Child1    | Top
 Child2    | Top
 Child3    | Top
 SubChild1 | Child1
 SubChild2 | Child1
```

```
(6 rows)

CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$
    SELECT name FROM nodes WHERE parent = $1
$$ LANGUAGE SQL STABLE;

SELECT * FROM listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, child FROM nodes, LATERAL listchildren(name) AS child;
  name  |   child
--------+-----------
 Top    | Child1
 Top    | Child2
 Top    | Child3
 Child1 | SubChild1
 Child1 | SubChild2
(5 rows)
```

В этом примере не делается ничего такого, что мы не могли бы сделать, применив простое соединение, но для более сложных вычислений возможность поместить некоторую логику в функцию может быть весьма удобной.

Функции, возвращающие множества, могут также вызываться в списке выборки запроса. Для каждой строки, которая генерируется самим запросом, вызывается функция, возвращающая множество, и для каждого элемента набора её результатов генерируется отдельная строка. Предыдущий пример можно было бы также переписать с применением запросов следующим образом:

```
SELECT listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, listchildren(name) FROM nodes;
  name  | listchildren
--------+--------------
 Top    | Child1
 Top    | Child2
 Top    | Child3
 Child1 | SubChild1
 Child1 | SubChild2
(5 rows)
```

Заметьте, что в последней команде `SELECT` для `Child2`, `Child3` и т. д. строки не выдаются. Это происходит потому, что `listchildren` возвращает пустое множество для этих аргументов, так что строки результата не генерируются. Это же поведение мы получаем при внутреннем соединении с результатом функции с применением `LATERAL`.

Поведение Postgres Pro с функциями, возвращающими множества, в списке выборки запроса практически не отличается от поведения с такими функциями, помещёнными в предложение `LATERAL FROM`. Например, запрос:

```
SELECT x, generate_series(1,5) AS g FROM tab;
```

почти равнозначен

```
SELECT x, g FROM tab, LATERAL generate_series(1,5) AS g;
```

Он мог быть полностью идентичным, но в данном конкретном примере планировщик может решить перенести `g` во внешнюю сторону соединения, так как `g` не имеет фактической зависимости по времени вычисления от `tab`. Такое решение привело бы к изменению порядка строк. Функции, возвращающие множества, в списке выборки всегда вычисляются так, как они вычислялись бы внутри соединения с вложенным циклом с остальным предложением `FROM`, так что эти функции выполняются до завершения прежде чем начинается рассмотрение следующей строки из предложения `FROM`.

Если в списке выборки запроса используются несколько функций, возвращающих запросы, они вычисляются примерно так же, как если бы они были помещены в один элемент `LATERAL ROWS FROM( ... )` предложения `FROM`. Для каждой строки из нижележащего запроса выдаётся строка с первым результатом каждой функции, а затем строка со вторым результатом и так далее. Если какие-либо из функций, возвращающих множества, выдают меньше результатов, чем другие, то вместо недостающих данных подставляются значения NULL, так что общее число строк, выдаваемых для одной нижележащей строки, равно числу строк, которое выдаёт функция с наибольшим количеством строк в возвращаемом множестве. Таким образом, функции, возвращающие множества, выполняются совместно, пока все их множества не будут исчерпаны, а затем выполнение продолжается со следующей нижележащей строкой.

Функции, возвращающие множества, могут быть вложенными в списке выборки, но это не допускается в элементах предложения `FROM`. В таких случаях каждый уровень вложенности обрабатывается отдельно, как если бы это был отдельный элемент `LATERAL ROWS FROM( ... )`. Например, в

```
SELECT srf1(srf2(x), srf3(y)), srf4(srf5(z)) FROM tab;
```

возвращающие множества функции `srf2`, `srf3` и `srf5` будут выполняться совместно для каждой строки `tab`, а затем `srf1` и `srf4` будут совместно применяться к каждой строке, произведённой нижними функциями.

Функции, возвращающие множества, нельзя использовать в конструкциях, вычисляемых по условию, например, `CASE` или `COALESCE`. Например, рассмотрите запрос

```
SELECT x, CASE WHEN x > 0 THEN generate_series(1, 5) ELSE 0 END FROM tab;
```

Может показаться, что он должен выдать пять экземпляров входных строк, в которых `x > 0`, и по одному экземпляру остальных строк; но на деле, так как `generate_series(1, 5)` будет выполняться в неявном элементе `LATERAL FROM` до того, как выражение `CASE` вообще будет рассматриваться, должно было бы выдаваться пять экземпляров абсолютно всех выходных строк. Во избежание путаницы в таких случаях выдаётся ошибка при разборе запроса.

Примечание

Если последняя команда функции — `INSERT`, `UPDATE` или `DELETE` с `RETURNING`, эта команда будет всегда выполняться до завершения, даже если функция не объявлена с указанием `SETOF` или вызывающий запрос не выбирает все строки результата. Все дополнительные строки, выданные предложением `RETURNING`, просто игнорируются, но соответствующие изменения в таблице всё равно произойдут (и будут завершены до выхода из функции).

Примечание

В Postgres Pro до версии 10 при помещении нескольких функций, возвращающих множества, в один список выборки поведение было не очень разумным, если они

возвращали не одинаковое число строк. В таких случаях число выходных строк равнялось наименьшему общему множителю количеств строк, возвращаемых этими функциями. Также и вложенные функции, возвращающие множества, работали не так, как описано выше; у такой функции мог быть максимум один аргумент, возвращающий множество, и каждая вложенность вычислялась независимо. Кроме того, ранее допускалось и условное выполнение (вычисление таких функций внутри `CASE` и т. п.), что ещё больше всё усложняло. При написании запросов, которые должны работать и со старыми версиями Postgres Pro, рекомендуется использовать синтаксис `LATERAL`, так как это гарантирует одинаковый результат с разными версиями. Если в вашем запросе используется условное вычисление функции, возвращающей множество, его можно исправить, переместив проверку условия в специально созданную функцию, возвращающую множество. Например:

```
SELECT x, CASE WHEN y > 0 THEN generate_series(1, z) ELSE 5 END FROM tab;
```

можно заменить на

```
CREATE FUNCTION case_generate_series(cond bool, start int, fin int, els int)
  RETURNS SETOF int AS $$
BEGIN
  IF cond THEN
    RETURN QUERY SELECT generate_series(start, fin);
  ELSE
    RETURN QUERY SELECT els;
  END IF;
END$$ LANGUAGE plpgsql;

SELECT x, case_generate_series(y > 0, 1, z, 5) FROM tab;
```

Это будет работать одинаково во всех версиях Postgres Pro.

35.4.9. Функции SQL, возвращающие таблицы (`TABLE`)

Есть ещё один способ объявить функцию, возвращающую множества, — использовать синтаксис `RETURNS TABLE(`*столбцы*`)`. Это равнозначно использованию одного или нескольких параметров `OUT` с объявлением функции как возвращающей `SETOF record` (или `SETOF` тип единственного параметра, если это применимо). Этот синтаксис описан в последних версиях стандарта SQL, так что этот вариант может быть более портируемым, чем `SETOF`.

Например, предыдущий пример с суммой и произведением можно также переписать так:

```
CREATE FUNCTION sum_n_product_with_tab (x int)
RETURNS TABLE(sum int, product int) AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;
```

Запись `RETURNS TABLE` не позволяет явно указывать `OUT` и `INOUT` для параметров — все выходные столбцы необходимо записать в списке `TABLE`.

35.4.10. Полиморфные функции SQL

Функции SQL могут быть объявлены как принимающие и возвращающие полиморфные типы `anyelement`, `anyarray`, `anynonarray`, `anyenum` и `anyrange`. За более подробным объяснением полиморфизма функций обратитесь к Подразделу 35.2.5. В следующем примере полиморфная функция `make_array` создаёт массив из двух элементов произвольных типов:

```
CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;
 intarray | textarray
```

```
----------+-----------
 {1,2}    | {a,b}
(1 row)
```

Обратите внимание на приведение типа `'a'::text`, определяющее, что аргумент имеет тип `text`. Оно необходимо, если аргумент задаётся просто строковой константой, так как иначе он будет воспринят как имеющий тип `unknown`, а массив типов `unknown` является недопустимым. Без этого приведения вы получите такую ошибку:

```
ERROR:  could not determine polymorphic type because input has type "unknown"
```

(ОШИБКА: не удалось определить полиморфный тип, так как входные аргументы имеют тип "unknown")

Функция с полиморфными аргументами может иметь фиксированный тип результата, однако обратное не допускается. Например:

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);
 is_greater
------------
 f
(1 row)

CREATE FUNCTION invalid_func() RETURNS anyelement AS $$
    SELECT 1;
$$ LANGUAGE SQL;
ERROR:  cannot determine result data type
DETAIL:  A function returning a polymorphic type must have at least one polymorphic
 argument.
```

(ОШИБКА: не удалось определить тип результата; ПОДРОБНОСТИ: Функция, возвращающая полиморфный тип, должна иметь минимум один полиморфный аргумент.)

Полиморфизм можно применять и с функциями, имеющими выходные аргументы. Например:

```
CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE SQL;

SELECT * FROM dup(22);
 f2 |   f3
----+---------
 22 | {22,22}
(1 row)
```

Полиморфизм также можно применять с функциями с переменными параметрами. Например:

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT anyleast(10, -1, 5, 4);
 anyleast
----------
       -1
(1 row)
```

```
SELECT anyleast('abc'::text, 'def');
 anyleast
----------
 abc
(1 row)

CREATE FUNCTION concat_values(text, VARIADIC anyarray) RETURNS text AS $$
    SELECT array_to_string($2, $1);
$$ LANGUAGE SQL;

SELECT concat_values('|', 1, 4, 2);
 concat_values
---------------
 1|4|2
(1 row)
```

35.4.11. Функции SQL с правилами сортировки

Когда функция SQL принимает один или несколько параметров сортируемых типов данных, правило сортировки определяется при каждом вызове функции, в зависимости от правил сортировки, связанных с фактическими аргументами, как описано в Разделе 22.2. Если правило сортировки определено успешно (то есть не возникло конфликтов между неявно установленными правилами сортировки аргументов), оно неявно назначается для всех сортируемых параметров. Выбранное правило будет определять поведение операций, связанных с сортировкой, в данной функции. Например, для показанной выше функции `anyleast`, результат

```
SELECT anyleast('abc'::text, 'ABC');
```

будет зависеть от правила сортировки по умолчанию, заданного в базе данных. С локалью `C` результатом будет строка `ABC`, но со многими другими локалями это будет `abc`. Нужное правило сортировки можно установить принудительно, добавив предложение `COLLATE` к одному из аргументов функции, например:

```
SELECT anyleast('abc'::text, 'ABC' COLLATE "C");
```

С другой стороны, если вы хотите, чтобы функция работала с определённым правилом сортировки, вне зависимости от того, с каким она была вызвана, вставьте предложения `COLLATE` где требуется в определении функции. Эта версия `anyleast` всегда будет сравнивать строки по правилам локали `en_US`:

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i] COLLATE "en_US") FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;
```

Но заметьте, что при попытке применить правило к несортируемому типу данных, возникнет ошибка.

Если для фактических аргументов не удаётся определить общее правило сортировки, функция SQL считает, что им назначено правило сортировки по умолчанию для их типа данных (обычно это то же правило сортировки, что определено по умолчанию для базы данных, но оно может быть и другим для параметров доменных типов).

Поведение сортируемых параметров можно воспринимать как ограниченную форму полиморфизма, применимую только к текстовым типам данных.

35.5. Перегрузка функций

Вы можете определить несколько функций с одним именем SQL, если эти функции будут принимать разные аргументы. Другими словами, имена функций можно *перегружать*. Независимо от того, используете вы эту возможность или нет, она требует предосторожности при вызове функций в базах данных, где одни пользователи не доверяют другим; см. Раздел 10.3. Когда выполняется запрос, сервер определяет, какую именно функцию вызывать, по количеству

и типам представленных аргументов. Перегрузка может быть полезна для имитации функций с переменным количеством аргументов, до какого-то конечного числа.

Создавая семейство перегруженных функций, необходимо не допускать неоднозначности. Например, если созданы функции:

```
CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

не вполне понятно, какая функция будет вызвана с довольно простыми аргументами вроде `test(1, 1.5)`. Реализованные в данный момент правила разрешения типов описаны в Главе 10, но разрабатывать систему, которая будет незаметно полагаться на такие особенности, неразумно.

Функции, принимающей один аргумент составного типа, обычно не следует давать имя, совпадающее с именем какого-либо атрибута (поля) этого типа. Вспомните, что запись *атрибут*(*таблица*) считается равнозначной *таблица*.*атрибут*. В случае, когда возникает неоднозначность между функцией, принимающей составной тип, и атрибутом составного типа, всегда будет выбираться атрибут. Этот выбор можно переопределить, дополнив имя функции схемой (то есть, записав *схема*.*функция*(*таблица*)), но лучше избежать этой проблемы, подобрав разные имена.

Другой тип конфликта возможен между обычными функциями и функциями с переменными параметрами. Например, можно создать функции `foo(numeric)` и `foo(VARIADIC numeric[])`. В этом случае будет непонятно, какая функция должна выбираться при передаче одного числового аргумента, например `foo(10.1)`. При разрешении этого конфликта предпочтение отдаётся функции, найденной первой по пути поиска, либо, если две функции находятся в одной схеме, выбирается функция с постоянными аргументами.

При перегрузке функций на языке C есть дополнительное ограничение: имя уровня C каждой функции в семействе перегруженных функций должно отличаться от имён уровня C всех других функций, как внутренних, так и загружаемых динамически. Если это правило нарушается, поведение зависит от среды. Вы можете получить ошибку компоновщика во время выполнения, либо будет вызвана не та функция (обычно внутренняя). Альтернативная форма предложения `AS` для SQL-команды `CREATE FUNCTION` позволяет отвязать имя SQL-функции от имени, определённого в исходном коде на C. Например:

```
CREATE FUNCTION test(int) RETURNS int
    AS 'имя_файла', 'test_1arg'
    LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
    AS 'имя_файла', 'test_2arg'
    LANGUAGE C;
```

Имена функций на C здесь следуют одному из множества возможных соглашений.

35.6. Категории изменчивости функций

Для каждой функции определяется характеристика *изменчивости*, с возможными вариантами: `VOLATILE`, `STABLE` и `IMMUTABLE`. Если эта характеристика не задаётся явно в команде CREATE FUNCTION, по умолчанию подразумевается `VOLATILE`. Категория изменчивости представляет собой обещание некоторого поведения функции для оптимизатора:

- Изменчивая функция (`VOLATILE`) может делать всё, что угодно, в том числе, модифицировать базу данных. Она может возвращать различные результаты при нескольких вызовах с одинаковыми аргументами. Оптимизатор не делает никаких предположений о поведении таких функций. В запросе, использующем изменчивую функцию, она будет вычисляться заново для каждой строки, когда потребуется её результат.
- Стабильная функция (`STABLE`) не может модифицировать базу данных и гарантированно возвращает одинаковый результат, получая одинаковые аргументы, для всех строк в одном операторе. Эта характеристика позволяет оптимизатору заменить множество вызовов этой функции одним. В частности, выражение, содержащее такую функцию, можно безопасно

использовать в условии поиска по индексу. (Так как при поиске по индексу целевое значение вычисляется только один раз, а не для каждой строки, использовать функцию с характеристикой VOLATILE в условии поиска по индексу нельзя.)

- Постоянная функция (IMMUTABLE) не может модифицировать базу данных и гарантированно всегда возвращает одинаковые результаты для одних и тех же аргументов. Эта характеристика позволяет оптимизатору предварительно вычислить функцию, когда она вызывается в запросе с постоянными аргументами. Например, запрос вида SELECT ... WHERE x = 2 + 2 можно упростить до SELECT ... WHERE x = 4, так как нижележащая функция оператора сложения помечена как IMMUTABLE.

Для наилучших результатов оптимизации, функции следует назначать самую строгую характеристику изменчивости, которой она соответствует.

Любая функция с побочными эффектами *должна* быть помечена как VOLATILE, чтобы обращения к ней не исключались при оптимизации. Даже если функция не имеет побочных эффектов, её нужно пометить как VOLATILE, если её значение может меняться при выполнении одного запроса; таковы функции random(), currval() и timeofday().

Другой важный пример представляет семейство функций current_timestamp, которые имеют характеристику STABLE, потому что их значения не меняются в рамках одной транзакции.

Характеристики STABLE и IMMUTABLE мало различаются, когда речь идёт о простых интерактивных запросах, которые планируются и сразу же выполняются; не имеет большого значения, будет ли функция выполнена однократно на этапе планирования или в начале выполнения. Существенное различие проявляется, когда план сохраняется и многократно используется позже. Если функция помечена как IMMUTABLE, тогда как на самом деле она не является постоянной, она может быть сведена к константе во время планирования, так что при последующих выполнениях плана вместо неё будет использоваться неактуальное значение. Это опасно при использовании подготовленных операторов или языков функций, кеширующих планы (например, PL/pgSQL).

У функций, написанных на SQL или на любом другом стандартном процедурном языке, есть ещё одно важное свойство, определяемое характеристикой изменчивости, а именно видимость изменений, произведённых командой SQL, которая вызывает эту функцию. Функция VOLATILE будет видеть такие изменения, тогда как STABLE и IMMUTABLE — нет. Это поведение реализуется посредством снимков в MVCC (см. Главу 13): STABLE и IMMUTABLE используют снимок, полученный в начале вызывающего запроса, тогда как функции VOLATILE получают свежий снимок в начале каждого запроса, который они выполняют.

> **Примечание**
>
> Функции, написанные на C, могут работать со снимками как угодно, но обычно лучше сделать так, чтобы они действовали аналогично.

Вследствие такой организации работы со снимками, функцию, содержащую только команды SELECT, можно безопасно пометить как STABLE, даже если она выбирает данные из таблиц, которые могут быть изменены параллельными запросами. Postgres Pro выполнит все команды в функции STABLE со снимком, полученным для вызывающего запроса, так что они будут видеть одно представление базы данных на протяжении всего запроса.

То же самое поведение со снимками распространяется на команды SELECT в функциях IMMUTABLE. Вообще в функциях IMMUTABLE обычно неразумно выбирать данные из таблиц, так как «постоянство» функции будет нарушено, если содержимое таблиц изменится. Однако Postgres Pro не принуждает вас явно отказаться от этого.

Одна из распространённых ошибок — помечать функцию как IMMUTABLE, при том, что её результаты зависят от параметра конфигурации. Например, функция, работающая с временем, может выдавать результаты, зависящие от параметра TimeZone. Для надёжности такие функции следует помечать как STABLE.

> **Примечание**
>
> Postgres Pro требует, чтобы функции `STABLE` и `IMMUTABLE` не содержали SQL-команд, кроме `SELECT`, для предотвращения модификации данных. (Это не совсем непробиваемое ограничение, так как эти функции всё же могут вызывать функции `VOLATILE`, способные модифицировать базу данных. Если вы реализуете такую схему, вы увидите, что функция `STABLE` и `IMMUTABLE` не замечает изменений в базе данных, произведённых вызванной функцией, так как они не проявляются в её снимке данных.)

35.7. Функции на процедурных языках

Postgres Pro позволяет разрабатывать собственные функции и на языках, отличных от SQL и C. Эти другие языки в целом обычно называются *процедурными языками* (PL, Procedural Languages). Процедурные языки не встроены в сервер Postgres Pro; они предлагаются загружаемыми модулями. За дополнительной информацией обратитесь к Главе 39 и следующим главам.

35.8. Внутренние функции

Внутренние функции — это функции, написанные на языке C, и статически скомпонованные в исполняемый код сервера Postgres Pro. В «теле» определения функции задаётся имя функции на уровне C, которое не обязательно должно совпадать с именем, объявленным для использования в SQL. (Обратной совместимости ради, тело функции может быть пустым, что будет означать, что имя функции на уровне C совпадает с именем в SQL.)

Обычно все внутренние функции, представленные на сервере, объявляются в ходе инициализации кластера баз данных (см. Раздел 17.2), но пользователь может воспользоваться командой `CREATE FUNCTION` и добавить дополнительные псевдонимы для внутренней функции. Внутренние функции объявляются в `CREATE FUNCTION` с именем языка `internal`. Например, так можно создать псевдоним для функции `sqrt`:

```
CREATE FUNCTION square_root(double precision) RETURNS double precision
    AS 'dsqrt'
    LANGUAGE internal
    STRICT;
```

(Большинство внутренних функций должны объявляться как «strict».)

> **Примечание**
>
> Не все «предопределённые» функции являются «внутренними» в вышеописанном смысле. Некоторые предопределённые функции написаны на SQL.

35.9. Функции на языке C

Пользовательские функции могут быть написаны на C (или на языке, который может быть совместим с C, например C++). Такие функции компилируются в динамически загружаемые объекты (также называемые разделяемыми библиотеками) и загружаются сервером по требованию. Именно метод динамической загрузки отличает функции «на языке C» от «внутренних» функций — правила написания кода по сути одни и те же. (Собственно, поэтому стандартная библиотека внутренних функций может быть богатым источником примеров для написания собственных функций на языке C.)

В настоящее время для функций на C применяется только одно соглашение о вызовах («версии 1»). Поддержка этого соглашения обозначается объявлением функции с макросом `(PG_FUNCTION_INFO_V1)`, как показано ниже.

35.9.1. Динамическая загрузка

В первый раз, когда в сеансе вызывается пользовательская функция в определённом внешнем объектном файле, загрузчик динамических модулей загружает этот файл в память, чтобы можно было вызвать эту функцию. Таким образом, в команде `CREATE FUNCTION`, объявляющей пользовательскую функцию на языке C, необходимо определить две сущности для функции: имя загружаемого объектного файла и имя уровня C (символ для компоновки) заданной функции в этом объектном файле. Если имя уровня C не указано явно, предполагается, что оно совпадает с именем функции в SQL.

Для нахождения разделяемого объектного файла по имени, заданному в команде `CREATE FUNCTION`, применяется следующий алгоритм:

1. Если имя задаётся абсолютным путём, загружается заданный файл.
2. Если имя начинается со строки `$libdir`, эта часть пути заменяется путём к каталогу библиотек Postgres Pro, который определяется во время сборки.
3. Если в имени не указывается каталог, поиск файла производится по пути, заданному конфигурационной переменной dynamic_library_path.
4. В противном случае (файл не был найден в пути поиска, или в его имени указывается не абсолютный путь к каталогу), загрузчик попытается принять имя как есть, что, скорее всего, не увенчается успехом. (Полагаться на текущий рабочий каталог ненадёжно.)

Если эта последовательность не даёт положительный результат, к данному имени добавляется принятое на данной платформе расширение файлов библиотек (часто `.so`) и последовательность повторяется снова. Если и это не приводит к успеху, происходит сбой загрузки.

Для поиска разделяемых библиотек рекомендуется задавать либо путь относительно `$libdir`, либо путь динамических библиотек. Это упрощает обновление версии при перемещении новой инсталляции в другое место. Какой именно каталог подразумевается под `$libdir`, можно узнать с помощью команды `pg_config --pkglibdir`.

Пользователь, от имени которого работает сервер Postgres Pro, должен иметь возможность пройти путь к файлу, который требуется загрузить. Очень распространённая ошибка — когда сам файл или каталог верхнего уровня оказывается недоступным для чтения и/или исполнения для пользователя postgres.

В любом случае имя файла, заданное в команде `CREATE FUNCTION`, записывается в системные каталоги буквально, так что если этот файл потребуется загрузить ещё раз, та же процедура будет проделана снова.

> **Примечание**
>
> Postgres Pro не будет компилировать функцию на C автоматически, поэтому прежде чем ссылаться на объектный файл в команде `CREATE FUNCTION`, его нужно скомпилировать. За дополнительными сведениями обратитесь к Подразделу 35.9.5.

Чтобы гарантировать, что динамически загружаемый объектный файл не будет загружен несовместимым сервером, Postgres Pro проверяет, содержит ли этот файл «отличительный блок» с требуемым содержимым Благодаря этому сервер может выявить очевидную несовместимость, например, когда код скомпилирован для другой старшей версии Postgres Pro. Отличительный блок стал обязательным, начиная с версии PostgreSQL 8.2. Чтобы включить его в свой модуль, напишите это в одном (и только одном) из исходных файлов модуля, после включения заголовочного файла `fmgr.h`:

```
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif
```

Проверку `#ifdef` можно опустить, если этот код не планируется компилировать для версий PostgreSQL до 8.2.

После того как он был использован первый раз, динамически загружаемый объектный файл сохраняется в памяти. Следующие обращения в том же сеансе к функциям в этом файле повлекут только небольшие издержки, связанные с поиском в таблице символов. Если вам нужно принудительно перезагрузить объектный файл, например, после перекомпиляции, начните новый сеанс.

Динамически загружаемый файл может дополнительно содержать функции инициализации и завершения работы библиотеки. Если в файле находится функция с именем `_PG_init`, эта функция будет вызвана сразу после загрузки файла. Эта функция не принимает параметры и не должна ничего возвращать. Если в файле находится функция `_PG_fini`, эта функция будет вызвана непосредственно перед выгрузкой файла. Эта функция так же не принимает параметры и не должна ничего возвращать. Заметьте, что `_PG_fini` будет вызываться только при выгрузке файла, но не при завершении процесса. (В настоящее время выгрузка отключена и не происходит никогда, но в будущем это может измениться.)

35.9.2. Базовые типы в функциях на языке C

Чтобы понимать, как написать функцию на языке C, вы должны знать, как внутри Postgres Pro представляются базовые типы данных и как их могут принимать и передавать функции. Postgres Pro внутри воспринимает базовые типы как «блоки памяти». Пользовательские функции, устанавливаемые для типов, в свою очередь, определяют, как Postgres Pro может работать с этими типами. То есть, Postgres Pro только сохраняет и загружает данные с диска, а для ввода, обработки и вывода данных он использует определяемые вами функции.

Базовые типы могут иметь один из трёх внутренних форматов:

- передаётся по значению, фиксированной длины
- передаётся по ссылке, фиксированной длины
- передаётся по ссылке, переменной длины

Типы, передаваемые по значению, могут иметь размер только 1, 2 или 4 байта (и 8 байт, если `sizeof(Datum)` равен 8 на вашей машине). Определяя собственные типы, следует позаботиться о том, чтобы они имели одинаковый размер (в байтах) во всех архитектурах. Например, тип `long` опасен, так как он имеет размер 4 байта на одних машинах, и 8 байт на других, тогда как тип `int` состоит из 4 байт в большинстве систем Unix. Поэтому разумной реализацией типа `int4` на платформе Unix может быть такая:

```
/* 4-байтное целое, передаётся по значению */
typedef int int4;
```

(В коде собственно Postgres Pro этот тип называется `int32`, так как в C принято соглашение, что `int`*XX* подразумевает *XX бит*. Заметьте, что вследствие этого тип `int8` в C имеет размер 1 байт. Тип `int8`, принятый в SQL, в C называется `int64`. См. также Таблицу 35.1.)

С другой стороны, типы фиксированной длины любого размера можно передавать по ссылке. Например, взгляните на пример реализации типа Postgres Pro:

```
/* 16-байтная структура, передаётся по ссылке */
typedef struct
{
    double  x, y;
} Point;
```

В функции Postgres Pro и из них могут передаваться только указатели на такие типы. Чтобы вернуть значение такого типа, выделите для него нужное количество памяти функцией `palloc`, заполните выделенную память и верните указатель на неё. (Если вы захотите просто вернуть то же

значение, что было получено во входном аргументе этого же типа данных, вы можете пропустить дополнительный вызов palloc и просто вернуть указатель на это поступившее значение.)

Наконец, все типы переменной длины также должны передаваться по ссылке. Все типы переменной длины должны начинаться с обязательного поля длины размером ровно 4 байта, которая будет задаваться макросом SET_VARSIZE; никогда не устанавливайте это поле вручную! Все данные, которые будут храниться в этом типе, должны размещаться в памяти непосредственно за этим полем длины. Поле длины содержит полную длину структуры, то есть включает размер самого поля длины.

Ещё один важный момент — старайтесь не оставлять неинициализированных байт в значениях данных; например, обнуляйте все возможные байты выравнивания, которые могут присутствовать в структурах. Если этого не делать, логически равные значения ваших данных могут представляться неравными планировщику, что приведёт к построению неэффективных (хотя и корректных) планов.

> **Предупреждение**
>
> *Никогда* не изменяйте содержимое, передаваемое на вход по ссылке. Если вы сделаете это, вы скорее всего испортите данные на диске, так как полученный вами указатель указывает непосредственно на место в дисковом буфере. Единственное исключение из этого правила освещается в Разделе 35.10.

В качестве примера мы можем определить тип text так:

```
typedef struct {
    int32 length;
    char data[FLEXIBLE_ARRAY_MEMBER];
} text;
```

Запись [FLEXIBLE_ARRAY_MEMBER] означает, что действительная длина массива данных в этом объявлении не указывается.

Работая с типами переменной длины, мы должны аккуратно выделить нужный объём памяти и записать его размер в поле длины. Например, если нужно сохранить 40 байт в структуре text, можно применить такой код:

```
#include "postgres.h"
...
char buffer[40]; /* our source data */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
SET_VARSIZE(destination, VARHDRSZ + 40);
memcpy(destination->data, buffer, 40);
...
```

VARHDRSZ совпадает с sizeof(int32), но для получения размера заголовка типа переменной длины хорошим стилем считается применять макрос VARHDRSZ. Кроме того, поле длины *должно* устанавливаться макросом SET_VARSIZE, а не простым присваиванием.

В Таблице 35.1 указано, какие типы языка C соответствуют типам SQL при написании функций на C с использованием встроенных типов Postgres Pro. В столбце «Определён в» указывается, какой заголовочный файл необходимо подключить, чтобы получить определение типа. (Фактическое определение может быть в другом файле, который подключается из указанного, однако рекомендуется придерживаться обозначенного интерфейса.) Заметьте, что в любом исходном файле первым всегда необходимо включать postgres.h, потому что в нём объявляется ряд вещей, которые нужны в любом случае, и потому что включение первым другого заголовочного файла может сделать код непереносимым.

Таблица 35.1. Типы C, эквивалентные встроенным типам SQL

Тип SQL	Тип C	Определён в
abstime	AbsoluteTime	utils/nabstime.h
boolean	bool	postgres.h (может быть встроен в компиляторе)
box	BOX*	utils/geo_decls.h
bytea	bytea*	postgres.h
"char"	char	(встроен в компиляторе)
character	BpChar*	postgres.h
cid	CommandId	postgres.h
date	DateADT	utils/date.h
float4 (real)	float4	postgres.h
float8 (double precision)	float8	postgres.h
int2 (smallint)	int16	postgres.h
int4 (integer)	int32	postgres.h
int8 (bigint)	int64	postgres.h
interval	Interval*	datatype/timestamp.h
lseg	LSEG*	utils/geo_decls.h
name	Name	postgres.h
numeric	Numeric	utils/numeric.h
oid	Oid	postgres.h
oidvector	oidvector*	postgres.h
path	PATH*	utils/geo_decls.h
point	POINT*	utils/geo_decls.h
regproc	RegProcedure	postgres.h
reltime	RelativeTime	utils/nabstime.h
text	text*	postgres.h
tid	ItemPointer	storage/itemptr.h
time	TimeADT	utils/date.h
time with time zone	TimeTzADT	utils/date.h
timestamp	Timestamp	datatype/timestamp.h
timestamp with time zone	TimestampTz	datatype/timestamp.h
tinterval	TimeInterval	utils/nabstime.h
varchar	VarChar*	postgres.h
xid	TransactionId	postgres.h

Теперь, когда мы рассмотрели все возможные структуры для базовых типов, мы можем перейти к примерам реальных функций.

35.9.3. Соглашение о вызовах версии 1

Соглашение о вызовах версии 1 полагается на макросы, скрывающие основную долю сложностей, связанных с передачей аргументов и результатов. По соглашению версии 1 функция на C должна всегда определяться так:

```
Datum funcname(PG_FUNCTION_ARGS)
```

В дополнение к этому, в том же исходном файле должен присутствовать вызов макроса:

```
PG_FUNCTION_INFO_V1(funcname);
```

(Обычно его принято записывать непосредственно перед функцией.) Этот вызов макроса не нужен для функций `internal`, так как Postgres Pro предполагает, что все внутренние функции используют соглашении версии 1. Однако для функций, загружаемых динамически, этот макрос необходим.

В функции версии 1 каждый аргумент выбирается макросом `PG_GETARG_`*`xxx`*`()`, который соответствует типу данных аргумента. В нестрогих функциях этому вызову должна предшествовать проверка на NULL в аргументе с использованием `PG_ARGISNULL()` (см. ниже). Результат возвращается макросом `PG_RETURN_`*`xxx`*`()` для возвращаемого типа. `PG_GETARG_`*`xxx`*`()` принимает в качестве параметра номер выбираемого аргумента функции (нумерация начинается с 0). `PG_RETURN_`*`xxx`*`()` принимает фактическое значение, которое нужно возвратить.

Несколько примеров использования соглашения о вызовах версии 1:

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"
#include "utils/geo_decls.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

/* by value */

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32   arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}

/* by reference, fixed length */

PG_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(PG_FUNCTION_ARGS)
{
    /* The macros for FLOAT8 hide its pass-by-reference nature. */
    float8   arg = PG_GETARG_FLOAT8(0);

    PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{
    /* Here, the pass-by-reference nature of Point is not hidden. */
    Point     *pointx = PG_GETARG_POINT_P(0);
    Point     *pointy = PG_GETARG_POINT_P(1);
```

```
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}

/* by reference, variable length */

PG_FUNCTION_INFO_V1(copytext);

Datum
copytext(PG_FUNCTION_ARGS)
{
    text     *t = PG_GETARG_TEXT_PP(0);

    /*
     * VARSIZE_ANY_EXHDR is the size of the struct in bytes, minus the
     * VARHDRSZ or VARHDRSZ_SHORT of its header.  Construct the copy with a
     * full-length header.
     */
    text     *new_t = (text *) palloc(VARSIZE_ANY_EXHDR(t) + VARHDRSZ);
    SET_VARSIZE(new_t, VARSIZE_ANY_EXHDR(t) + VARHDRSZ);

    /*
     * VARDATA is a pointer to the data region of the new struct.  The source
     * could be a short datum, so retrieve its data through VARDATA_ANY.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA_ANY(t), /* source */
           VARSIZE_ANY_EXHDR(t));   /* how many bytes */
    PG_RETURN_TEXT_P(new_t);
}

PG_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(PG_FUNCTION_ARGS)
{
    text  *arg1 = PG_GETARG_TEXT_PP(0);
    text  *arg2 = PG_GETARG_TEXT_PP(1);
    int32 arg1_size = VARSIZE_ANY_EXHDR(arg1);
    int32 arg2_size = VARSIZE_ANY_EXHDR(arg2);
    int32 new_text_size = arg1_size + arg2_size + VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA_ANY(arg1), arg1_size);
    memcpy(VARDATA(new_text) + arg1_size, VARDATA_ANY(arg2), arg2_size);
    PG_RETURN_TEXT_P(new_text);
}
```

В предположении, что приведённый выше код был подготовлен в файле `funcs.c` и скомпилирован в разделяемый объект, мы можем объявить эти функции в Postgres Pro следующими командами:

```
CREATE FUNCTION add_one(integer) RETURNS integer
     AS 'КАТАЛОГ/funcs', 'add_one'
```

```
     LANGUAGE C STRICT;

-- обратите внимание — это перегрузка SQL-функции "add_one"
CREATE FUNCTION add_one(double precision) RETURNS double precision
     AS 'КАТАЛОГ/funcs', 'add_one_float8'
     LANGUAGE C STRICT;

CREATE FUNCTION makepoint(point, point) RETURNS point
     AS 'КАТАЛОГ/funcs', 'makepoint'
     LANGUAGE C STRICT;

CREATE FUNCTION copytext(text) RETURNS text
     AS 'КАТАЛОГ/funcs', 'copytext'
     LANGUAGE C STRICT;

CREATE FUNCTION concat_text(text, text) RETURNS text
     AS 'КАТАЛОГ/funcs', 'concat_text'
     LANGUAGE C STRICT;
```

Здесь *КАТАЛОГ* — это путь к каталогу, в который помещён разделяемый библиотечный файл (например, каталог учебных материалов (tutorial) в исходном коде Postgres Pro, содержащий код примеров, использованных в этом разделе). (Лучше было бы просто написать `'funcs'` в предложении `AS`, предварительно добавив *КАТАЛОГ* в путь поиска. В любом случае мы можем опустить принятое в системе расширение файлов разделяемых библиотек, обычно `.so`.)

Заметьте, что мы объявили эти функции как «strict» (строгие) — это означает, что система будет автоматически подразумевать результат NULL, если в одном из входных значений передаётся NULL. Благодаря этому, мы избегаем необходимости проверять входные значения на NULL в коде функции. Без такого объявления нам пришлось бы явно проверять параметры на NULL, используя `PG_ARGISNULL()`.

Макрос `PG_ARGISNULL(`*n*`)` позволяет функции проверить на NULL каждый из её аргументов. (Разумеется, это нужно делать только в функциях, объявленных без характеристики «strict».) Как и с макросом `PG_GETARG_`*xxx*`()`, входные аргументы нумеруются, начиная с нуля. Заметьте, что не следует обращаться к макросу `PG_GETARG_`*xxx*`()`, не убедившись, что соответствующий аргумент не NULL. Чтобы возвратить NULL в качестве результата, воспользуйтесь макросом `PG_RETURN_NULL()`; это работает и со строгими, и с нестрогими функциями.

На первый взгляд соглашения о вызовах версии 1 могут показаться всего лишь бессмысленным мракобесием, по сравнению с соглашениями простого `C`. Однако они позволяют работать с аргументами и возвращаемыми значениями, в которых может передаваться `NULL`, а также со значениями в формате TOAST (сжатыми или хранимыми отдельно).

Кроме того, в интерфейсе версии 1 появились две вариации макроса `PG_GETARG_`*xxx*`()`. Первая вариация, `PG_GETARG_`*xxx*`_COPY()`, гарантированно возвращает копию указанного аргумента, которую можно безопасно модифицировать. (Обычный макрос иногда возвращает указатель на значение, которое физически хранится в таблице, в которую нельзя писать. С макросом `PG_GETARG_`*xxx*`_COPY()` гарантированно получается результат, доступный для записи.) Вторая вариация представлена макросом `PG_GETARG_`*xxx*`_SLICE()`, принимающим три параметра. В первом передаётся номер аргумента функции (как и раньше). Во втором и третьем передаётся смещение и длина сегмента, который должен быть возвращён. Смещение отсчитывается с нуля, а отрицательная длина указывает, что запрашивается оставшаяся часть значения. Эти макросы дают более эффективный доступ к частям больших значений, имеющим тип хранения «external». (Тип хранения столбца может задаваться командой `ALTER TABLE` *имя_таблицы* `ALTER COLUMN` *имя_столбца* `SET STORAGE` *тип_хранения*, где *тип_хранения*: `plain`, `external`, `extended` или `main`.)

Наконец, соглашения о вызовах версии 1 позволяют возвращать множества (Подраздел 35.9.8) и реализовывать триггерные функции (Глава 36) и обработчики вызовов процедурных языков (Глава 51).

35.9.4. Написание кода

Прежде чем перейти к более сложным темам, мы должны обсудить некоторые правила написания кода функций на языке C для Postgres Pro. Хотя принципиально можно загружать в Postgres Pro функции, написанные на языках, отличных от C, обычно это довольно сложно (когда вообще возможно), так как другие языки, например C++, FORTRAN или Pascal часто не следуют соглашениям, принятым в C. То есть другие языки могут передавать аргументы и возвращаемые значения между функциями разными способами. Поэтому далее предполагается, что ваши функции на языке C действительно написаны на C.

Основные правила написания и компиляции функций на C таковы:

- Чтобы выяснить, где находятся заголовочные файлы сервера Postgres Pro, установленные в вашей системе (или в системе, с которой будут работать ваши пользователи), воспользуйтесь командой `pg_config --includedir-server`.
- Для компиляции и компоновки кода, который можно будет динамически загрузить в Postgres Pro, требуется указать специальные флаги. Чтобы конкретнее узнать, как это сделать в вашей конкретной операционной системе, обратитесь к Подразделу 35.9.5.
- Не забудьте определить «отличительный блок» для вашей разделяемой библиотеки, как описано в Подразделе 35.9.1.
- Для выделения памяти используйте функцию Postgres Pro `palloc`, а для освобождения `pfree`, вместо соответствующих функций библиотеки C `malloc` и `free`. Память, выделяемая функцией `palloc`, будет автоматически освобождаться в конце каждой транзакции, во избежание утечек памяти.
- Всегда обнуляйте байты ваших структур, применяя `memset` (или сразу выделяйте память функцией `palloc0`). Даже если вы присвоите значение каждому полю структуры, в ней могут оставаться байты выравнивания (пустоты в структуре), содержащие случайные значения. Если исключить это требование, будет сложно поддерживать индексы или соединение по хешу, так как для вычисления хеша придётся выбирать только значащие биты из вашей структуры данных. Планировщик также иногда полагается на побитовое сравнение констант, так что результаты планирования могут оказаться неожиданными, если логически равные значения окажутся неравными на битовом уровне.
- Большинство внутренних типов Postgres Pro объявлены в `postgres.h`, тогда как интерфейс менеджера функций (`PG_FUNCTION_ARGS` и т. д.) определён в `fmgr.h`, так что потребуется подключить как минимум два этих файла. По соображениям портируемости, лучше включить `postgres.h` *первым*, до каких-либо других системных или пользовательских файлов заголовков. При подключении `postgres.h` автоматически также будут подключены `elog.h` и `palloc.h`.
- Имена символов, определённые в объектных файлах, не должны конфликтовать друг с другом или с именами других символов, определённых в исполняемых файлах сервера Postgres Pro. Если вы столкнётесь с ошибками, вызванными таким конфликтом, вам придётся переименовать ваши функции или переменные.

35.9.5. Компиляция и компоновка динамически загружаемых функций

Прежде чем вы сможете использовать ваши написанные на C функции, расширяющие возможности Postgres Pro, их необходимо скомпилировать и скомпоновать особым образом, чтобы сервер мог динамически загрузить полученный файл. Точнее говоря, вам необходимо создать *разделяемую библиотеку*.

За подробной информацией, дополняющей и поясняющей то, что описано в этом разделе, вам следует обратиться к документации вашей операционной системы, в частности, к страницам руководства компилятора C, `cc`, и компоновщика, `ld`. Кроме того, ряд рабочих примеров можно найти в каталоге `contrib` исходного кода Postgres Pro. Однако если вы непосредственно воспользуйтесь этими примерами, ваши модули окажутся зависимыми от наличия исходного кода Postgres Pro.

Создание разделяемых библиотек в принципе не отличается от сборки исполняемых файлов: сначала исходные файлы компилируются в объектные, а затем объектные связываются вместе. Объектные файлы должны создаваться так, чтобы они содержали *позиционно-независимый* код (PIC, position-independent code), что означает, что при загрузке для выполнения этот код может быть помещён в любое место в памяти. (Объектные файлы, предназначенные для сборки непосредственно исполняемых файлов, обычно собираются не так.) Команда для компоновки разделяемой библиотеки принимает специальные флаги, что отличают её от компоновки исполняемого файла (по крайней мере в теории — в некоторых системах реальность не так прекрасна).

В следующих примерах предполагается, что исходный код находится в файле `foo.c` и мы будем создавать разделяемую библиотеку `foo.so`. Промежуточный объектный файл будет называться `foo.o`, если не отмечено другое. Разделяемая библиотека может включать больше одного объектного файла, но здесь мы ограничимся одним.

FreeBSD

Для создания кода PIC компилятору передаётся флаг `-fPIC`. Чтобы создать разделяемую библиотеку, используется флаг компилятора `-shared`.

```
gcc -fPIC -c foo.c
gcc -shared -o foo.so foo.o
```

Это применимо как минимум к FreeBSD версии 3.0.

HP-UX

Для создания кода PIC системному компилятору передаётся флаг `+z`, а компилятору GCC — флаг `-fPIC`. Чтобы создать разделяемые библиотеки, используется флаг компоновщика `-b`. Таким образом, нужно выполнить:

```
cc +z -c foo.c
```

или:

```
gcc -fPIC -c foo.c
```

а затем:

```
ld -b -o foo.sl foo.o
```

В HP-UX, в отличие от многих других систем, для разделяемых библиотек выбрано расширение `.sl`.

Linux

Для создания кода PIC компилятору передаётся флаг `-fPIC`. Для создания разделяемой библиотеки компилятору передаётся флаг `-shared`. Полный пример будет выглядеть так:

```
cc -fPIC -c foo.c
cc -shared -o foo.so foo.o
```

macOS

Следующий пример показывает нужные команды, в предположении, что установлены инструменты разработчика.

```
cc -c foo.c
cc -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```

NetBSD

Для создания кода PIC компилятору передаётся флаг `-fPIC`. Для компоновки разделяемых библиотек в системах ELF компилятору передаётся флаг `-shared`, а в старых системах, не поддерживающих ELF, применяется команда `ld -Bshareable`.

```
gcc -fPIC -c foo.c
gcc -shared -o foo.so foo.o
```

OpenBSD

Для создания кода PIC компилятору передаётся флаг `-fPIC`, а для компоновки разделяемых библиотек применяется команда `ld -Bshareable`.

```
gcc -fPIC -c foo.c
ld -Bshareable -o foo.so foo.o
```

Solaris

Для создания кода PIC компилятору Sun передаётся флаг `-KPIC`, а компилятору GCC — флаг `-fPIC`. Для компоновки разделяемой библиотеки можно передать обоим компиляторам флаг `-G` либо передать флаг `-shared` компилятору GCC.

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o
```

или

```
gcc -fPIC -c foo.c
gcc -G -o foo.so foo.o
```

> **Подсказка**
>
> Если это слишком сложно для вас, попробуйте использовать средство *GNU Libtool*, которое скрывает различия платформ за единым интерфейсом.

Полученную разделяемую библиотеку можно будет затем загрузить в Postgres Pro. Когда команде `CREATE FUNCTION` передаётся имя файла, это должно быть имя файла разделяемой библиотеки, а не промежуточного объектного файла. Заметьте, что принятое в системе расширение файлов библиотек (как правило, `.so` или `.sl`) в команде `CREATE FUNCTION` можно опустить, и так обычно следует делать для лучшей портируемости.

Чтобы уточнить, где сервер будет искать файлы разделяемых библиотек, вернитесь к Подразделу 35.9.1.

35.9.6. Аргументы составного типа

Составные типы не имеют фиксированного макета данных, как структуры C. В частности, экземпляры составного типа могут содержать поля NULL. Кроме того, в контексте наследования составные типы могут иметь разные поля для разных членов в одной иерархии наследования. Поэтому Postgres Pro предоставляет функциям специальный интерфейс для обращения к полям составных типов из C.

Предположим, что мы хотим написать функцию, отвечающую на запрос:

```
SELECT name, c_overpaid(emp, 1500) AS overpaid
    FROM emp
    WHERE name = 'Bill' OR name = 'Sam';
```

С соглашениями о вызовах версии 1 мы можем определить функцию `c_overpaid` так:

```
#include "postgres.h"
#include "executor/executor.h"  /* for GetAttributeByName() */

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(c_overpaid);

Datum
c_overpaid(PG_FUNCTION_ARGS)
{
```

```
    HeapTupleHeader  t = PG_GETARG_HEAPTUPLEHEADER(0);
    int32            limit = PG_GETARG_INT32(1);
    bool isnull;
    Datum salary;

    salary = GetAttributeByName(t, "salary", &isnull);
    if (isnull)
        PG_RETURN_BOOL(false);
    /* Alternatively, we might prefer to do PG_RETURN_NULL() for null salary. */

    PG_RETURN_BOOL(DatumGetInt32(salary) > limit);
}
```

`GetAttributeByName` — это системная функция Postgres Pro, которая возвращает атрибуты указанной строки. Она принимает три параметра: аргумент типа `HeapTupleHeader`, имя нужного атрибута и выходной параметр, устанавливаемый, если значение атрибута — NULL. `GetAttributeByName` возвращает значение `Datum`, которое вы можете привести к подходящему типу данных, используя соответствующий макрос `DatumGet`*`XXX`*`()`. Заметьте, что возвращаемое значение недействительно, если установлен флаг null; всегда проверяйте этот флаг, прежде чем что-либо делать с результатом.

Есть также функция `GetAttributeByNum`, которая выбирает целевой атрибут не по имени, а по номеру столбца.

Следующая команда объявляет функцию `c_overpaid` в SQL:

```
CREATE FUNCTION c_overpaid(emp, integer) RETURNS boolean
    AS 'КАТАЛОГ/funcs', 'c_overpaid'
    LANGUAGE C STRICT;
```

Заметьте, что мы использовали `STRICT`, чтобы нам не пришлось проверять входные аргументы на равенство NULL.

35.9.7. Возврат строк (составных типов)

Чтобы вернуть строку или значение составного типа из функции на языке C, можно использовать специальный API, предоставляющий макросы и функции, скрывающие основную сложность формирования составных типов данных. Для использования этого API необходимо включить в исходный файл:

```
#include "funcapi.h"
```

Сформировать значение составного типа (далее «кортеж») можно двумя способами: его можно построить из массива значений Datum, или из массива строк C, которые будут переданы функциям преобразования ввода для типов столбцов кортежа. В любом случае сначала нужно получить или сконструировать дескриптор `TupleDesc` для структуры кортежа. Работая со значениями Datum, вы передаёте `TupleDesc` функции `BlessTupleDesc`, а затем вызываете `heap_form_tuple` для каждой строки. Работая со строками C, вы передаёте `TupleDesc` функции `TupleDescGetAttInMetadata`, а затем для каждой строки вызываете `BuildTupleFromCStrings`. В случае функции, возвращающей множество кортежей, все подготовительные действия можно выполнить один раз при первом вызове функции.

Для получения требуемого дескриптора `TupleDesc` предлагается несколько дополнительных функций. Рекомендованный способ возврата составных значений заключается в вызове функции:

```
TypeFuncClass get_call_result_type(FunctionCallInfo fcinfo,
                                   Oid *resultTypeId,
                                   TupleDesc *resultTupleDesc)
```

При этом в `fcinfo` должна передаваться та же структура, что была передана самой вызывающей функции. (Для этого, конечно, необходимо использовать соглашения о вызовах версии 1.) В

resultTypeId можно передать NULL или адрес локальной переменной, в которую будет записан OID типа результата функции. В resultTupleDesc должен передаваться адрес локальной переменной TupleDesc. Убедить, что функция возвратила результат TYPEFUNC_COMPOSITE; в этом случае в resultTupleDesc оказывается требуемая структура TupleDesc. (Если получен другой результат, вы можете выдать ошибку с сообщением «функция, возвращающая запись, вызвана в контексте, не допускающем этот тип».)

> **Подсказка**
>
> get_call_result_type позволяет получить фактический тип результата полиморфной функции, так что она полезна и в функциях, возвращающих скалярные полиморфные результаты, не только в функциях, возвращающих составные типы. Выходной параметр resultTypeId полезен в первую очередь для полиморфных скалярных функций.

> **Примечание**
>
> В дополнение к get_call_result_type есть схожая функция get_expr_result_type, позволяющая получить ожидаемый тип результата для вызова функции, представленного деревом выражения. Её можно использовать, когда тип результата нужно определить извне самой функции. Есть также функция get_func_result_type, которую можно применять, когда известен только OID функции. Однако эти две функции неспособны выдать тип результата функций, возвращающих record, а get_func_result_type неспособна разрешать полиморфные типы, так что вместо них лучше использовать get_call_result_type.

Ранее для получения TupleDesc использовались теперь уже устаревшие функции:

```
TupleDesc RelationNameGetTupleDesc(const char *relname)
```

(возвращает TupleDesc для типа строк указанного отношения) и:

```
TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)
```

(возвращает TupleDesc для типа, задаваемого по OID). Применяя её, можно получить TupleDesc для базового или составного типа. Однако она не подойдёт для функции, возвращающей тип record, и не сможет разрешить полиморфные типы.

Получив TupleDesc, вызовите:

```
TupleDesc BlessTupleDesc(TupleDesc tupdesc)
```

если вы планируете работать со структурами Datum, либо:

```
AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)
```

если планируете работать со строками C. Если вы разрабатываете функцию, возвращающую набор данных, вы можете сохранить результаты этих функций в структуре FuncCallContext, в поле tuple_desc или attinmeta, соответственно.

Если вы работаете со структурами Datum, воспользуйтесь функцией:

```
HeapTuple heap_form_tuple(TupleDesc tupdesc, Datum *values, bool *isnull)
```

Она формирует HeapTuple из переданных ей данных в форме Datum.

Если вы работаете со строками C, воспользуйтесь функцией:

```
HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char **values)
```

Она формирует HeapTuple из переданных ей данных в виде строк C. В параметре *values* ей передаётся массив строк C, по одной для каждого атрибута выходной строки. Каждая из этих строк должна иметь формат, принимаемый функцией ввода типа данных атрибута. Чтобы задать значение NULL для одного из этих атрибутов, вместо соответствующего указателя в массиве *values* нужно передать NULL. Эту функцию нужно вызывать для каждой строки, которую вы будете возвращать.

Получив кортеж, который вы будете возвращать из вашей функции, вы должны преобразовать его в тип `Datum`. Чтобы преобразовать `HeapTuple` в Datum, воспользуйтесь функцией:

```
HeapTupleGetDatum(HeapTuple tuple)
```

Полученный тип `Datum` можно вернуть непосредственно, если должна возвращаться только одна строка, либо использовать как текущее выдаваемое значение в функции, возвращающей набор строк.

Пример приведён в следующем разделе.

35.9.8. Возврат множеств

Функции на языке C могут возвращать наборы данных (множества строк) двумя способами. Первый способ, который называется *ValuePerCall* (значение за вызов), заключается в многократном вызове функции (при этом ей каждый раз передаются одни и те же аргументы). Эта функция при очередном вызове должна возвращать следующую строку, пока не выдаст все строки, о чём она сообщает, возвращая NULL. Таким образом, возвращающая множество функция (Set-Returning Function, SRF) должна сохранять между вызовами своё состояние в достаточном объёме, чтобы помнить, какие данные уже были выданы, и возвращать следующие при очередном вызове. Второй вариант, *Materialize* (Материализация), заключается в формировании в SRF объекта tuplestore, содержащего сразу весь результирующий набор; единственный вызов производится для получения сразу всего результата, и никакое состояние между вызовами сохранять не нужно.

Реализуя режим ValuePerCall, важно не забывать, что выполнение запроса до полного завершения не гарантируется. Так, например, получив указание `LIMIT`, исполнитель запроса может перестать вызывать возвращающую множество функцию до получения всех строк. Это означает, что выполнять действия, связанные с очисткой, в последнем вызове небезопасно, так как он может вовсе не состояться. Поэтому для функций, которым нужно обращаться к внешним ресурсам, например, открывая файловые дескрипторы, рекомендуется использовать режим материализации результатов.

В продолжении этого раздела описываются несколько вспомогательных макросов, которые часто используются (хотя не являются обязательными) в SRF, реализующих метод ValuePerCall.

Для использования описанных здесь макросов поддержки ValuePerCall подключите `funcapi.h`. Эти макросы работают со структурой `FuncCallContext`, содержащей состояние, которое требуется сохранять между вызовами. Указатель на `FuncCallContext` внутри вызываемой SRF сохраняется между вызовами в поле `fcinfo->flinfo->fn_extra`, которое макросы автоматически заполняют при первом использовании, рассчитывая прочесть из него тот же указатель при последующих вызовах.

```
typedef struct FuncCallContext
{
    /*
     * Счётчик числа ранее выполненных вызовов
     *
     * call_cntr сбрасывается в 0 макросом SRF_FIRSTCALL_INIT() и
     * увеличивается на 1 каждый раз, когда вызывается SRF_RETURN_NEXT().
     */
    uint64 call_cntr;

    /*
     * Максимальное число вызовов (может не использоваться)
     *
     * max_calls не является обязательным и присутствует здесь только для удобства.
     * Если это значение не задано, вы должны предоставить другую возможность
 определить,
     * когда функция завершила свою работу.
     */
```

```
    uint64 max_calls;

    /*
     * Указатель на слот результата (может не использоваться)
     *
     * Это поле устарело и представлено только для обратной совместимости, а именно,
     * для пользовательских SRF, использующих устаревшую TupleDescGetSlot().
     */
    TupleTableSlot *slot;

    /*
     * Указатель на разнообразную контекстную информацию,
     * представленную пользователем; (может не использоваться)
     *
     * user_fctx используется как указатель на ваши собственные данные,
     * позволяющий сохранить контекстную информацию между вызовами функции.
     */
    void *user_fctx;

    /*
     * Указатель на структуру, содержащую метаданные ввода типа атрибута
     * (может не использоваться)
     *
     * attinmeta задействуется, когда возвращаются кортежи (т. е. составные типы
данных),
     * и не применяется для возврата базовых типов. Он нужен, только если
     * вы планируете использовать BuildTupleFromCStrings() для формирования
возвращаемого
     * кортежа.
     */
    AttInMetadata *attinmeta;

    /*
     * Контекст памяти, нужный для структур, которые должны сохраняться при нескольких
вызовах
     *
     * Поле multi_call_memory_ctx заполняется в SRF_FIRSTCALL_INIT() и используется
     * в SRF_RETURN_DONE() для очистки. Это наиболее подходящий контекст
     * для любых блоков памяти, которые должны многократно использоваться при
     * повторных вызовах SRF.
     */
    MemoryContext multi_call_memory_ctx;

    /*
     * Указатель на структуру, содержащую описание кортежа (может не использоваться)
     *
     * tuple_desc задействуется, когда возвращаются кортежи (т. е. составные типы),
     * и нужен только, если вы планируете формировать кортежи с помощью функции
     * heap_form_tuple(), а не BuildTupleFromCStrings().  Заметьте, что сохраняемый
     * здесь указатель TupleDesc обычно должен сначала пройти через вызов
     * BlessTupleDesc().
     */
    TupleDesc tuple_desc;

} FuncCallContext;
```

Для SRF предоставляется ряд макросов, использующих эту инфраструктуру:

```
SRF_IS_FIRSTCALL()
```

Используйте этот макрос, чтобы определить, вызывается ли ваша функция в первый раз. При первом вызове (но не при последующих) выполните:

```
SRF_FIRSTCALL_INIT()
```

для того, чтобы инициализировать `FuncCallContext`. При каждом вызове функции, включая первый, выполняйте:

```
SRF_PERCALL_SETUP()
```

для того, чтобы подготовиться к использованию `FuncCallContext`.

Если у вашей функции есть данные, которые она должна выдать в текущем вызове, выполните:

```
SRF_RETURN_NEXT(funcctx, result)
```

для того, чтобы передать их вызывающему. (Переменная `result` должна быть типа `Datum`, либо одним значением, либо кортежем, подготовленным как описано выше.) Наконец, когда ваша функция закончила выдавать данные, выполните:

```
SRF_RETURN_DONE(funcctx)
```

для того, чтобы провести очистку и завершить SRF.

Контекст памяти, в котором вызывается SRF, временный, он будет очищаться между вызовами. Это значит, что вам не нужно вызывать `pfree` для всех блоков памяти, которые вы получили через `palloc`; они всё равно будут освобождены. Однако если вы хотите выделить структуры данных, сохраняющиеся между вызовами, вам нужно разместить их где-то в другом месте. Для размещения данных, которые не должны уничтожаться, пока SRF не закончит работу, подходит контекст памяти, на который указывает `multi_call_memory_ctx`. В большинстве случаев это означает, что вы должны переключиться в контекст `multi_call_memory_ctx` в коде подготовки при первом вызове. Для сохранения указателя на такие долгоживущие структуры воспользуйтесь полем `funcctx->user_fctx`. (Память, которую вы получаете в контексте `multi_call_memory_ctx`, будет освобождена автоматически при завершении запроса, так что и её освобождать вручную нет необходимости.)

> **Предупреждение**
>
> Тогда как фактические аргументы такой функции не меняются от вызова к вызову, если вы распаковываете значения аргументов (что обычно прозрачно делают макросы `PG_GETARG_`*`xxx`*) во временном контексте, распакованные копии будут освобождаться при каждом вызове. Соответственно, если вы сохраните ссылки на такие значения в своём контексте `user_fctx`, вы должны либо скопировать эти значения в `multi_call_memory_ctx` после распаковки, либо распаковывать значения только в этом контексте.

Полный пример с псевдокодом будет выглядеть так:

```
Datum
my_set_returning_function(PG_FUNCTION_ARGS)
{
    FuncCallContext  *funcctx;
    Datum             result;
    другие необходимые объявления

    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext oldcontext;

        funcctx = SRF_FIRSTCALL_INIT();
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);
        /* Код подготовки при первом вызове: */
        некоторый код
        если возвращается составной тип
```

```
            получить TupleDesc и, возможно, AttInMetadata
        конец ветвления для составного типа
        некоторый код
        MemoryContextSwitchTo(oldcontext);
    }

    /* Код подготовки для каждого вызова: */
    некоторый код
    funcctx = SRF_PERCALL_SETUP();
    некоторый код

    /* Только так мы можем определить, не последний ли это вызов: */
    if (funcctx->call_cntr < funcctx->max_calls)
    {
        /* Здесь мы возвращаем ещё один результат: */
        некоторый код
        получение результирующих данных
        SRF_RETURN_NEXT(funcctx, result);
    }
    else
    {
        /* Мы заканчиваем выдавать результаты, и это надо отразить. */
        /* (Воздержитесь от соблазна написать здесь код, освобождающий ресурсы.) */
        SRF_RETURN_DONE(funcctx);
    }
}
```

Полный пример простой SRF-функции, возвращающей составной тип, выглядит так:

```
PG_FUNCTION_INFO_V1(retcomposite);

Datum
retcomposite(PG_FUNCTION_ARGS)
{
    FuncCallContext     *funcctx;
    int                  call_cntr;
    int                  max_calls;
    TupleDesc            tupdesc;
    AttInMetadata       *attinmeta;

    /* stuff done only on the first call of the function */
    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext   oldcontext;

        /* create a function context for cross-call persistence */
        funcctx = SRF_FIRSTCALL_INIT();

        /* switch to memory context appropriate for multiple function calls */
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);

        /* total number of tuples to be returned */
        funcctx->max_calls = PG_GETARG_UINT32(0);

        /* Build a tuple descriptor for our result type */
        if (get_call_result_type(fcinfo, NULL, &tupdesc) != TYPEFUNC_COMPOSITE)
            ereport(ERROR,
                    (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                     errmsg("function returning record called in context "
```

```
                            "that cannot accept type record")));

        /*
         * generate attribute metadata needed later to produce tuples from raw
         * C strings
         */
        attinmeta = TupleDescGetAttInMetadata(tupdesc);
        funcctx->attinmeta = attinmeta;

        MemoryContextSwitchTo(oldcontext);
    }

    /* stuff done on every call of the function */
    funcctx = SRF_PERCALL_SETUP();

    call_cntr = funcctx->call_cntr;
    max_calls = funcctx->max_calls;
    attinmeta = funcctx->attinmeta;

    if (call_cntr < max_calls)    /* do when there is more left to send */
    {
        char       **values;
        HeapTuple    tuple;
        Datum        result;

        /*
         * Prepare a values array for building the returned tuple.
         * This should be an array of C strings which will
         * be processed later by the type input functions.
         */
        values = (char **) palloc(3 * sizeof(char *));
        values[0] = (char *) palloc(16 * sizeof(char));
        values[1] = (char *) palloc(16 * sizeof(char));
        values[2] = (char *) palloc(16 * sizeof(char));

        snprintf(values[0], 16, "%d", 1 * PG_GETARG_INT32(1));
        snprintf(values[1], 16, "%d", 2 * PG_GETARG_INT32(1));
        snprintf(values[2], 16, "%d", 3 * PG_GETARG_INT32(1));

        /* build a tuple */
        tuple = BuildTupleFromCStrings(attinmeta, values);

        /* make the tuple into a datum */
        result = HeapTupleGetDatum(tuple);

        /* clean up (this is not really necessary) */
        pfree(values[0]);
        pfree(values[1]);
        pfree(values[2]);
        pfree(values);

        SRF_RETURN_NEXT(funcctx, result);
    }
    else    /* do when there is no more left */
    {
        SRF_RETURN_DONE(funcctx);
    }
}
```

В SQL её можно объявить следующим образом:

```
CREATE TYPE __retcomposite AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION retcomposite(integer, integer)
    RETURNS SETOF __retcomposite
    AS 'имя_файла', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

Также её можно объявить с параметрами OUT:

```
CREATE OR REPLACE FUNCTION retcomposite(IN integer, IN integer,
    OUT f1 integer, OUT f2 integer, OUT f3 integer)
    RETURNS SETOF record
    AS 'имя_файла', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

Заметьте, что при таком подходе выходным типом функции формально является анонимный тип `record`.

35.9.9. Полиморфные типы аргументов и результата

Функции на языке C могут быть объявлены как принимающие и возвращающие полиморфные типы `anyelement`, `anyarray`, `anynonarray`, `anyenum` и `anyrange`. За более подробным объяснением полиморфных функций обратитесь к Подразделу 35.2.5. Когда типы аргументов или результата определены как полиморфные, автор функции не может заранее знать, с какими типами данных она будет вызываться и какой возвращать. Чтобы функция на C в стиле версии 1 могла определить фактические типы данных своих аргументов и тип, который она должна вернуть, в `fmgr.h` предлагаются две функции. Они называются `get_fn_expr_rettype(FmgrInfo *flinfo)` и `get_fn_expr_argtype(FmgrInfo *flinfo, int argnum)` и возвращают соответственно OID типа результата и аргумента, либо `InvalidOid`, если информация о типе отсутствует. Структуру `flinfo` обычно можно получить по ссылке `fcinfo->flinfo`. Номер аргумента `argnum` задаётся, начиная с нуля. В качестве альтернативы `get_fn_expr_rettype` также можно использовать функции `get_call_result_type`. Кроме того, есть функция `get_fn_expr_variadic`, позволяющая определить, были ли переменные аргументы объединены в массив. Это полезно в основном для функций `VARIADIC "any"`, так как такое объединение всегда имеет место для функций с переменными аргументами, принимающих обычные типы.

Например, предположим, что нам нужно написать функцию, принимающую один элемент любого типа и возвращающую одномерный массив этого типа:

```
PG_FUNCTION_INFO_V1(make_array);
Datum
make_array(PG_FUNCTION_ARGS)
{
    ArrayType  *result;
    Oid         element_type = get_fn_expr_argtype(fcinfo->flinfo, 0);
    Datum       element;
    bool        isnull;
    int16       typlen;
    bool        typbyval;
    char        typalign;
    int         ndims;
    int         dims[MAXDIM];
    int         lbs[MAXDIM];

    if (!OidIsValid(element_type))
        elog(ERROR, "could not determine data type of input");
```

```
    /* получить переданный элемент, учитывая, что это может быть NULL */
    isnull = PG_ARGISNULL(0);
    if (isnull)
        element = (Datum) 0;
    else
        element = PG_GETARG_DATUM(0);

    /* мы имеем дело с одной размерностью */
    ndims = 1;
    /* и одним элементом */
    dims[0] = 1;
    /* с нижней границей, равной 1 */
    lbs[0] = 1;

    /* получить требуемую информацию о типе элемента */
    get_typlenbyvalalign(element_type, &typlen, &typbyval, &typalign);

    /* теперь создать массив */
    result = construct_md_array(&element, &isnull, ndims, dims, lbs,
                                element_type, typlen, typbyval, typalign);

    PG_RETURN_ARRAYTYPE_P(result);
}
```

Следующая команда объявляет функцию `make_array` в SQL:

```
CREATE FUNCTION make_array(anyelement) RETURNS anyarray
    AS 'КАТАЛОГ/funcs', 'make_array'
    LANGUAGE C IMMUTABLE;
```

Существует один вариант полиморфизма, которым могут пользоваться только функции на языке C: их можно объявить с параметрами типа `"any"`. (Заметьте, что имя этого типа нужно заключать в двойные кавычки, так как это также зарезервированное слово в SQL.) Он работает так же, как `anyelement`, за исключением того, что он не требует, чтобы аргументы `"any"` имели одинаковый тип, и не помогает определить тип результата функции. Функцию на языке C можно также объявить с последним параметром `VARIADIC "any"`. Ему будут соответствовать один или более фактических аргументов любого типа (не обязательно одинакового). Эти аргументы *не* будут собираться в массив, как это происходит с обычными функциями с переменными аргументами; они просто будут переданы функции по отдельности. Если применяется этот вариант, то чтобы определить число фактических аргументов и их типы, нужно использовать макрос `PG_NARGS()` и функции, описанные выше. Пользователи такой функции также могут пожелать использовать ключевое слово `VARIADIC` в вызове функции, ожидая, что функция обработает элементы массива как отдельные аргументы. При необходимости соответствующее поведение должна реализовывать сама функция, определив с помощью `get_fn_expr_variadic`, был ли фактический аргумент передан с указанием `VARIADIC`.

35.9.10. Функции преобразования

Некоторые вызовы функций можно упростить на стадии планирования, в зависимости от особых свойств функции. Например, функцию умножения (`int4mul(n, 1)`) можно свести просто к `n`. Чтобы определить такую оптимизацию, нужно написать *функцию преобразования* и поместить её OID в поле `protransform` записи основной функции в `pg_proc`. Функция преобразования должна иметь в SQL сигнатуру `protransform(internal) RETURNS internal`. В аргументе, фактически имеющем тип `FuncExpr *`, передаётся фиктивный узел, представляющий вызов основной функции. Если анализ дерева выражения в функции преобразования показывает, что вместо всех возможных конкретных вызовов может быть подставлено дерево упрощённого выражения, эта функция должна построить и вернуть это упрощённое выражение. В противном случае нужно вернуть указатель `NULL` (*не* NULL языка SQL).

Мы не гарантируем, что Postgres Pro никогда не будет вызывать основную функцию во всех случаях, когда функция преобразования может её упростить, поэтому важно, чтобы упрощённое выражение строго соответствовало реальному вызову основной функции.

В настоящее время эта функциональность не предоставляется пользователям на уровне SQL из соображений безопасности, так что на практике это можно использовать только для оптимизации встроенных функций.

35.9.11. Разделяемая память и лёгкие блокировки

Модули расширений могут резервировать лёгкие блокировки и область в разделяемой памяти при запуске сервера. Чтобы библиотека модуля предварительно загружалась на этапе запуска сервера, нужно указать её в shared_preload_libraries. Чтобы зарезервировать разделяемую память, вызовите из вашей функции `_PG_init` функцию:

```
void RequestAddinShmemSpace(int size)
```

Чтобы зарезервировать лёгкие блокировки, из `_PG_init` нужно вызвать:

```
void RequestNamedLWLockTranche(const char *tranche_name, int num_lwlocks)
```

В результате будет сформирован массив из `num_lwlocks` лёгких блокировок под именем `tranche_name`. Чтобы получить указатель на этот массив, воспользуйтесь функцией `GetNamedLWLockTranche`.

Во избежание возможных условий гонки каждый обслуживающий процесс должен вызывать `AddinShmemInitLock` в момент подключения и при инициализации разделяемой памяти, как показано здесь:

```
static mystruct *ptr = NULL;

if (!ptr)
{
        bool    found;

        LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
        ptr = ShmemInitStruct("my struct name", size, &found);
        if (!found)
        {
                /* инициализировать содержимое области разделяемой памяти; */
                /* получить все требуемые блокировки LWLocks:
 */                ptr->locks = GetNamedLWLockTranche("my tranche name");
        }
        LWLockRelease(AddinShmemInitLock);
}
```

35.9.12. Использование C++ для расширяемости

Хотя код сервера Postgres Pro написан на C, расширения для него можно писать и на C++, если соблюдать эти правила:

- Все функции, к которым будет обращаться сервер, должны предоставлять ему интерфейс C; эти функции на C затем могут вызывать функции на языке C++. В частности, для функций, доступных серверу, необходимо указать `extern C`. Это также необходимо для всех функций, указатели на которые передаются между кодом сервера и подключаемым кодом на C++.
- Освобождайте память, применяя для этого подходящий метод. Например, память сервера в основном выделяется функцией `palloc()`, так что освобождать её нужно, вызывая `pfree()`. Попытка использовать в таких случаях принятую в C++ операцию `delete` приведёт к ошибке.
- Не допускайте распространения исключений в код C (добавляйте блок, перехватывающий все исключения, на верхнем уровне функций `extern C`). Это необходимо, даже если код на C++ не генерирует исключения явно, потому что исключения могут возникать, например,

и при нехватке памяти. Все исключения должны перехватываться, и в интерфейс C должны передаваться соответствующие ошибки. Если возможно, скомпилируйте код C++ с указанием `-fno-exceptions`, чтобы полностью отключить исключения; в таких случаях вы должны будете выявлять исключительные ситуации в коде C++, например, проверять на NULL адрес, возвращённый `new()`.

- Вызывая серверные функции из кода C++, убедитесь, что в стеке вызова C++ содержатся только простые структуры данных. Это необходимо, потому что в случае ошибки сервера выполняется функция `longjmp()`, а она не отматывает стек вызовов C++ должным образом для объектов, отличных от простых структур.

Резюмируя, лучше всего поместить код C++ за ограду из функций `extern C`, которые будут доступны серверу и смогут защитить от исключений, а также потери стека вызовов и утечки памяти.

35.10. Пользовательские агрегатные функции

Агрегатные функции в Postgres Pro определяются в терминах *значений состояния* и *функций перехода состояния*. То есть агрегатная функция работает со значением состояния, которое меняется при обработке каждой последующей строки. Чтобы определить агрегатную функцию, нужно выбрать тип данных для значения состояния, начальное значение состояния и функцию перехода состояния. Функция перехода состояния принимает предыдущее значение состояния и входное агрегируемое значение для текущей строки и возвращает новое значение состояния. Также можно указать *функцию завершения*, на случай, если ожидаемый результат агрегатной функции отличается от данных, которые сохраняются в изменяющемся значении состояния. Функция завершения принимает конечное значение состояния и возвращает то, что она хочет вернуть в виде результата агрегирования. В принципе, функции перехода и завершения представляют собой просто обычные функции, которые также могут применяться вне контекста агрегирования. (На практике для большей производительности часто создаются специализированные функции перехода, которые работают, только когда вызываются при агрегировании.)

Таким образом, помимо типов данных аргументов и результата, с которыми имеет дело пользователь агрегатной функции, есть также тип данных внутреннего состояния, который может отличаться от этих типов.

Если мы определяем агрегат, не использующий функцию завершения, наш агрегат будет вычислять бегущее значение функции по столбцам каждой строки. Примером такой агрегатной функции является `sum`. Вычисление `sum` начинается с нуля, а затем к накапливаемой сумме всегда прибавляется значение из текущей строки. Например, если мы хотим сделать агрегатную функцию `sum` для комплексных чисел, нам потребуется только функция сложения для такого типа данных. Такая агрегатная функция может быть определена так:

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)'
);
```

Использовать её можно будет так:

```
SELECT sum(a) FROM test_complex;

   sum
-----------
 (34,53.9)
```

(Заметьте, что мы задействуем перегрузку функций: в системе есть несколько агрегатных функций с именем `sum`, но Postgres Pro может определить, какая именно из них применима к столбцу типа `complex`.)

Определённая выше функция `sum` вернёт ноль (начальное значение состояния), если в наборе данных не окажется значений, отличных от NULL. У нас может возникнуть желание вернуть NULL в этом случае — стандарт SQL требует, чтобы `sum` работала так. Мы можем добиться этого, просто опустив фразу `initcond`, так что начальным значением состояния будет NULL. Обычно это будет означать, что в `sfunc` придётся проверять входное значение состояния на NULL. Но для `sum` и некоторых других простых агрегатных функций, как `max` и `min`, достаточно вставить в переменную состояния первое входное значение не NULL, а затем начать применять функцию перехода со следующего значения не NULL. Postgres Pro сделает это автоматически, если начальное значение состояние равно NULL и функция перехода помечена как «strict» (то есть не должна вызываться для аргументов NULL).

Ещё одна особенность поведения по умолчанию «строгой» функции перехода — предыдущее значение состояния остаётся без изменений, когда встречается значение NULL. Другими словами, значения NULL игнорируются. Если вам нужно другое поведение для входных значений NULL, не объявляйте свою функцию перехода строгой (strict); вместо этого, проверьте в ней поступающие значения на NULL и обработайте их, как требуется.

Функция `avg` (вычисляющая среднее арифметическое) представляет собой более сложный пример агрегатной функции. Ей необходимы два компонента текущего состояния: сумма входных значений и их количество. Окончательный результат получается как частное этих величин. При реализации этой функции для значения состояния обычно используется массив. Например, встроенная реализация `avg(float8)` выглядит так:

```
CREATE AGGREGATE avg (float8)
(
    sfunc = float8_accum,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0,0}'
);
```

> **Примечание**
>
> Функция `float8_accum` принимает массив из трёх, а не двух элементов, так как в дополнение к количеству и сумме значений она подсчитывает ещё сумму их квадратов. Это сделано для того, чтобы её можно было применять для `avg` и для некоторых других агрегатных функций.

Вызовы агрегатных функций SQL допускают указания `DISTINCT` и `ORDER BY`, которые определяют, какие строки и в каком порядке будут поступать в функцию перехода агрегата. Это реализовано на заднем плане и непосредственно не затрагивает функции, поддерживающие работу агрегата.

За дополнительными подробностями обратитесь к описанию CREATE AGGREGATE.

35.10.1. Режим движущегося агрегата

Агрегатные функции могут дополнительно поддерживать *режим движущегося агрегата*, который позволяет значительно быстрее выполнять агрегатные функции в окнах со сдвигающимся началом рамки. (За информацией об использовании агрегатных функций в качестве оконных обратитесь к Разделу 3.5 и Подразделу 4.2.8.) Основная идея состоит в том, что помимо добавления обычной функции перехода «вперёд», для агрегатной функции задаётся *функция обратного перехода*, которая позволяет убирать строки из накапливаемого значения состояния, когда они покидают рамку окна. Например, для `sum` в качестве функции прямого перехода выполняется сложение, а в качестве функции обратного перехода выполняется вычитание. Без функции обратного перехода механизм оконных функций вынужден вычислять агрегат заново при каждом перемещении начала рамки, в результате чего время обработки оказывается пропорциональным количеству входных строк, помноженному на средний размер рамки. С функцией обратного перехода это время пропорционально только количеству входных строк.

Функции обратного перехода передаётся текущее значение состояния и агрегируемое входное значение(я) для строки, ранее учтённой в текущем состоянии. Она должна восстановить то значение состояния, которое было бы получено, если бы эта строка не агрегировалась, но агрегировались все последующие. Иногда для этого нужно, чтобы функция обратного перехода сохраняла больше информации о состоянии, чем это требуется для простого режима агрегирования. Таким образом, для режима движущегося агрегата используется реализация, отличная от простого режима: для него определяется отдельный тип данных, отдельная функция прямого перехода и отдельная функция завершения, при необходимости. Они могут совпадать с типом данных и аналогичными функциями обычного режима, если в дополнительном состоянии необходимости нет.

В качестве примера мы можем доработать показанную выше агрегатную функцию `sum`, чтобы она поддерживала режим движущегося агрегата так:

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)',
    msfunc = complex_add,
    minvfunc = complex_sub,
    mstype = complex,
    minitcond = '(0,0)'
);
```

Параметры, имена которых начинаются с `m`, определяют реализацию для движущегося агрегата. За исключением функции обратного перехода, `minvfunc`, они соответствуют параметрам обычного агрегата без `m`.

Функции прямого перехода в режиме движущегося агрегата не разрешено возвращать NULL в качестве нового значения состояния. Если функция обратного перехода возвращает NULL, это воспринимается как признак того, что она не может восстановить предыдущее состояние для полученных данных, и значит, агрегатное вычисление нужно производить заново с текущей позиции начала рамки. Это соглашение позволяет применять режим движущегося агрегата и в ситуациях, когда прокручивать назад значение состояния непрактично. Функция обратного перехода может «спасовать» в таких случаях, но включаться в работу, насколько это возможно в большинстве случаев. Например, агрегатная функция, работающая с числами с плавающей точкой, может спасовать, когда от неё потребуется убрать значение `NaN` (не число, not a number) из текущего значения состояния.

Разрабатывая функции, реализующие режим движущегося агрегата, важно, чтобы функция обратного перехода могла восстановить в точности требуемое значение состояния. Иначе в результатах могут проявляться различия в зависимости от того, использовался ли режим движущегося агрегата. Например, на первый взгляд может показаться, что легко добавить функцию обратного перехода для сложения, но заявленное требование не будет выполняться для `sum` с типом `float4` или `float8`. Наивное объявление `sum(float8)` может быть таким:

```
CREATE AGGREGATE unsafe_sum (float8)
(
    stype = float8,
    sfunc = float8pl,
    mstype = float8,
    msfunc = float8pl,
    minvfunc = float8mi
);
```

Однако такой агрегат может выдавать результаты, радикально отличающиеся от тех, что он выдавал бы без функции обратного перехода. Например, рассмотрите запрос

```
SELECT
  unsafe_sum(x) OVER (ORDER BY n ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING)
```

```
FROM (VALUES (1, 1.0e20::float8),
             (2, 1.0::float8)) AS v (n,x);
```

Он возвращает `0` в качестве второго результата, а не ожидаемое значение `1`. Это связано с ограниченной точностью значений с плавающей точкой: при добавлении `1` к `1e20` снова получается `1e20`, а при вычитании `1e20` из результата получается `0`, а не `1`. Заметьте, что это принципиальное ограничение арифметики чисел с плавающей точкой, а не недостаток Postgres Pro.

35.10.2. Агрегатные функции с полиморфными и переменными аргументами

Агрегатная функция может использовать полиморфные функции перехода состояния или функции завершения, так что эти функции могут применяться для реализации нескольких агрегатов. За объяснением полиморфных функций обратитесь к Подразделу 35.2.5. Более того, сама агрегатная функция может описываться с полиморфными типами входных данных и состояния, так что одно определение агрегатной функции может служить для использования с разными типами данных. Пример полиморфного агрегата:

```
CREATE AGGREGATE array_accum (anyelement)
(
    sfunc = array_append,
    stype = anyarray,
    initcond = '{}'
);
```

Здесь фактическим типом состояния для любого конкретного агрегатного вызова будет массив, элементы которого будут иметь тип входных данных. Действие данного агрегата заключается в накоплении всех входных значений в массиве этого типа. (К вашему сведению: встроенная агрегатная функция `array_agg` обеспечивает подобную функциональность, но работает быстрее, чем могла бы функция с приведённым определением.)

Так будут выглядеть результаты с аргументами двух различных типов:

```
SELECT attrelid::regclass, array_accum(attname)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
    GROUP BY attrelid;

   attrelid    |              array_accum
---------------+---------------------------------------
 pg_tablespace | {spcname,spcowner,spcacl,spcoptions}
(1 row)

SELECT attrelid::regclass, array_accum(atttypid::regtype)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
    GROUP BY attrelid;

   attrelid    |        array_accum
---------------+---------------------------
 pg_tablespace | {name,oid,aclitem[],text[]}
(1 row)
```

Обычно агрегатная функция с полиморфным типом результата имеет и полиморфный тип состояния, как в предыдущем примере. Это необходимо, так как иначе нельзя будет объявить функцию завершения: она должна будет иметь полиморфный тип результата, но не будет иметь полиморфного аргумента, что команда `CREATE FUNCTION` не примет на основании того, что тип результата нельзя будет определить при вызове. Но иметь полиморфный тип состояния не всегда удобно. Чаще всего эта проблема возникает, когда функции реализации агрегата пишутся на C и тип состояния должен объявляться как `internal`, так как для него нет

соответствующего типа на уровне SQL. Чтобы решить эту проблему, можно объявить функцию завершения как принимающую дополнительные фиктивные аргументы, соответствующие входным аргументам агрегата. В этих фиктивных аргументах всегда передаются значения NULL, так как при вызове функции завершения какое-либо определённое значение отсутствует. Единственное их предназначение — позволить связать тип результата полиморфной функции завершения с типом входных данных агрегата. Например, определение встроенного агрегата `array_agg` выглядит так:

```
CREATE FUNCTION array_agg_transfn(internal, anynonarray)
  RETURNS internal ...;
CREATE FUNCTION array_agg_finalfn(internal, anynonarray)
  RETURNS anyarray ...;

CREATE AGGREGATE array_agg (anynonarray)
(
    sfunc = array_agg_transfn,
    stype = internal,
    finalfunc = array_agg_finalfn,
    finalfunc_extra
);
```

Здесь параметр `finalfunc_extra` указывает, что функция завершения помимо значения состояния получит дополнительные фиктивные аргументы, соответствующие входным аргументам агрегата. Дополнительный аргумент `anynonarray` позволяет сделать объявление `array_agg_finalfn` допустимым.

Агрегатную функцию можно сделать принимающей переменное число аргументов, объявив её последний аргумент как массив `VARIADIC`, в том же ключе, как и обычную функцию; см. Подраздел 35.4.5. При этом у функций перехода агрегата их последний аргумент должен иметь тот же тип массива. Такие функции обычно также объявляются как `VARIADIC`, но строго это не требуется.

> **Примечание**
>
> Агрегатные функции с переменными аргументами легко допускают ошибочное использование в сочетании с указанием `ORDER BY` (см. Подраздел 4.2.7), так как анализатор запроса не может определить, было ли передано нужное количество фактических параметров в такой комбинации. Помните, что всё, находящееся справа от `ORDER BY`, является ключом сортировки, а не аргументом агрегатной функции. Например, в
>
> ```
> SELECT myaggregate(a ORDER BY a, b, c) FROM ...
> ```
>
> анализатор запроса увидит один агрегатный аргумент функции и три ключа сортировки. Однако пользователь мог подразумевать и следующее:
>
> ```
> SELECT myaggregate(a, b, c ORDER BY a) FROM ...
> ```
>
> Если функция `myaggregate` принимает переменные аргументы, оба эти вызова будут вполне допустимы.
>
> По этой же причине, стоит подумать дважды, прежде чем создавать агрегатные функции с одинаковыми именами, но разным числом обычных аргументов.

35.10.3. Сортирующие агрегатные функции

Описанные выше агрегатные функции были «обычными» агрегатами. Но Postgres Pro также поддерживает *сортирующие агрегатные функции*, которые имеют два отличия от обычных. Во-первых, в дополнение к обычным агрегируемым аргументам, вычисляемых для каждой входной строки, сортирующий агрегат может иметь «непосредственные» аргументы, которые должны вычисляться в операции агрегирования только один раз. Во-вторых, для обычных агрегируемых

аргументов порядок их сортировки задаётся явно, а сортирующий агрегат обычно выполняет вычисления, зависящие от конкретного порядка строк, например, вычисляет ранг или процентиль, так что порядок сортировки критичен для каждого вызова. Например, встроенное определение функции `percentile_disc` равнозначно следующему:

```
CREATE FUNCTION ordered_set_transition(internal, anyelement)
  RETURNS internal ...;
CREATE FUNCTION percentile_disc_final(internal, float8, anyelement)
  RETURNS anyelement ...;

CREATE AGGREGATE percentile_disc (float8 ORDER BY anyelement)
(
    sfunc = ordered_set_transition,
    stype = internal,
    finalfunc = percentile_disc_final,
    finalfunc_extra
);
```

Этот агрегат принимает непосредственный аргумент `float8` (дробь процентиля) и агрегируемые данные, которые могут быть любого упорядочиваемого типа. Используя его, можно рассчитать средний семейный доход следующим образом:

```
SELECT percentile_disc(0.5) WITHIN GROUP (ORDER BY income) FROM households;
 percentile_disc
-----------------
           50489
```

В данном случае `0.5` — это непосредственный аргумент; если бы дробь процентиля менялась от строки к строке, это не имело бы смысла.

В отличие от случая с обычными агрегатами, сортировка входных строк для сортирующего агрегата *не* выполняется на заднем плане, а является задачей функций, реализующих агрегат. Обычный подход к такой реализации заключается в сохранении ссылки на объект «tuplesort» в значении состояния агрегата, загрузке поступающих строк в этот объект, и собственно завершении сортировки и обработки данных в функции завершения. Такая конструкция позволяет функции завершения выполнять специальные операции, в частности, вставлять дополнительные «гипотетические» строки в сортируемые данные. Тогда как обычные агрегаты часто реализуются функциями, написанными на PL/pgSQL или другом процедурном языке, сортирующие агрегатные функции обычно должны быть написаны на C, так как их значение состояния нельзя выразить каким-либо типом данных SQL. (Обратите внимание, что в приведённом выше примере значение состояния объявлено как имеющее тип `internal` — это типичный вариант.)

Функция перехода состояния для сортирующего агрегата получает значение текущего состояния плюс агрегируемые входные данные для каждой строки и возвращает изменённое значение состояния. Это определение распространяется и на обычные агрегаты, но заметьте, что непосредственные аргументы (если они есть) при этом не передаются. Функция завершения же получает последнее значение состояния и значения непосредственных аргументов (если они есть), а также (если присутствует указание `finalfunc_extra`) значения NULL, соответствующие агрегируемым данным. С обычными агрегатами указание `finalfunc_extra` действительно полезно, только если агрегат полиморфный; тогда дополнительные фиктивные аргументы необходимы, чтобы связать тип результата функции завершения с типом(ами) входных данных агрегата.

В настоящее время сортирующие агрегаты не могут использоваться в качестве оконных функций, поэтому от них поддержка режима движущегося агрегата не требуется.

35.10.4. Частичное агрегирование

Дополнительно агрегатная функция может поддерживать *частичное агрегирование*. Идея такого агрегирования в том, чтобы вызывать функции перехода состояния для различных подмножеств

входных данных независимо, а затем комбинировать значения состояния, вычисленные по этим подмножествам, и получать тот же результат, что был бы получен при сканировании сразу всех входных данных. Этот режим может применяться для параллельного агрегирования, когда разные рабочие процессы сканируют различные части таблицы. При этом каждый рабочий процесс выдаёт частичное значение состояния, а в конце эти значения комбинируются вместе и получается окончательное значение состояния. (В будущем этот режим может также применяться, например для комбинированного агрегирования локальных и удалённых таблиц, но пока это не реализовано.)

Для поддержки частичного агрегирования в определении агрегатной функции должна задаваться *комбинирующая функция*, принимающая два значения типа состояния агрегата (представляющие результаты агрегирования по двум подмножествам входных строк) и выдающая новое значение типа состояния, представляющее то состояние, которое было бы получено при агрегировании совокупности этих подмножеств строк. При этом относительный порядок входных строк в этих двух множествах не оговаривается. Это значит, что для агрегатных функций, зависящих от порядка входных строк, обычно невозможно определить осмысленную комбинирующую функцию.

В качестве простого примера, частичное агрегирование могут поддерживать функции `MAX` и `MIN`, если задать в качестве комбинирующей соответственно функцию сравнения значений большее-из-двух или меньшее-из-двух, ту же, что они используют и как функцию перехода. Для `SUM` комбинирующей функцией будет просто функция сложения. (И это опять же функция перехода, если только значение состояния не выходит за рамки типа входных данных.)

Комбинирующая функция задействуется практически так же, как функция перехода, но принимает в качестве второго аргумента значение типа состояния, а не нижележащего входного типа. В частности, на неё распространяются те же правила строгости функции и передачи значений NULL. Также учтите, что если в определении агрегатной функции задаётся отличное от NULL значение `initcond`, оно будет задавать начальное состояние не только для каждого прохода частичного агрегирования, но также и начальное состояние для комбинирующей функции, которая будет вызываться для комбинирования каждого частичного результата в этом состоянии.

Если типом состояния агрегатной функции выбран `internal`, комбинирующая функция отвечает за то, чтобы её результат был помещён в контекст памяти, подходящий для значений агрегатного состояния. В частности это значит, что, получив в первом аргументе `NULL`, нельзя просто возвратить второй аргумент, так как это значение окажется в неверном контексте и не просуществует достаточное время.

Когда типом состояния агрегатной функции выбран `internal`, обычно в определении агрегатной функции также уместно задать *функцию сериализации* и *функцию десериализации*, которые позволяют копировать значение состояния из одного процесса в другой. Без этих функций параллельное агрегирование невозможно, а также вероятно не будут работать такие будущие приложения, как локальное/удалённое агрегирование.

Функция сериализации должна принимать один аргумент типа `internal` и возвращать результат типа `bytea`, представляющий значение состояния, упакованное в плоский набор байтов. Функция десериализации, напротив, обращает это преобразование. Она должна принимать два аргумента типов `bytea` и `internal` и возвращать результат типа `internal`. (Второй её аргумент не используется и всегда равен нулю, но он требуется из соображений типобезопасности.) Результат функции десериализации следует просто разместить в текущем контексте памяти, так как в отличие от результата комбинирующей функции он недолговечен.

Также стоит заметить, что для выполнения агрегатной функции в параллельном режиме она должна иметь характеристику `PARALLEL SAFE` (безопасная для распараллеливания). Характеристики допустимости распараллеливания её опорных функций значения не имеют.

35.10.5. Вспомогательные функции для агрегатов

Функция, написанная на C, может определить, была ли она вызвана как вспомогательная функция агрегирования, вызвав `AggCheckCallContext`, например, так:

```
if (AggCheckCallContext(fcinfo, NULL))
```

Смысл такой проверки в том, что для функции перехода (когда эта функция возвращает true) первым входным аргументом является временное значение состояния, которое можно безопасно модифицировать на месте, не создавая новую копию. Пример вы можете увидеть в функции `int8inc()`. (Это *единственный* случай, когда функция может безопасно изменять входные данные, передаваемые по ссылке. В частности, функции завершения для обычных агрегатов не должны изменять входные данные ни в коем случае, так как в некоторых ситуациях они могут вызываться повторно с тем же конечным значением состояния.)

Второй аргумент `AggCheckCallContext` можно использовать, чтобы получить контекст памяти, в котором содержатся значения агрегатного состояния. Это полезно для функций перехода, которые желают использовать «развёрнутые» объёкты (см. Подраздел 35.11.1) в качестве значений состояния. При первом вызове такая функция перехода должна возвратить развёрнутый объект в контексте памяти, относящемся к контексту состояния агрегата, а затем продолжать возвращать тот же объект при последующих вызовах. Например, эту логику можно увидеть в функции `array_append()`. (Функция `array_append()` не используется в качестве перехода никаким встроенным агрегатом, но она написана так, чтобы работать эффективно в таком качестве в дополнительном агрегате.)

Ещё одна вспомогательная подпрограмма, предназначенная для агрегатных функций, написанных на C, называется `AggGetAggref`. Эта функция возвращает узел разбора `Aggref`, описывающий вызов агрегата. Это в основном полезно для сортирующих агрегатов, которые могут исследовать структуру узла `Aggref` и выяснить, какой порядок сортировки они должны реализовать. Примеры использования можно найти в `orderedsetaggs.c` в исходном коде Postgres Pro.

35.11. Пользовательские типы

Как описывалось в Разделе 35.2, Postgres Pro может расширяться и поддерживать новые типы данных. В этом разделе описывается, как определить новые базовые типы, то есть типы данных, описанные ниже уровня языка SQL. Для создания нового базового типа необходимо реализовать функции, работающие с этим типом, на языке низкого уровня, обычно C.

Пользовательский тип должен всегда иметь функции ввода и вывода. Эти функции определяют, как тип будет выглядеть в строковом виде (при вводе и выводе для пользователя) и как этот тип размещается в памяти. Функция ввода принимает в качестве аргумента строку символов, заканчивающуюся нулём, и возвращает внутреннее представление типа (в памяти). Функция вывода принимает в качестве аргумента внутреннее представление типа и возвращает строку символов, заканчивающуюся нулём. Если мы хотим не просто сохранить тип, но делать с ним нечто большее, мы должны предоставить дополнительные функции, реализующие все операции, которые мы хотели бы иметь для этого типа.

Предположим, что нам нужен тип `complex`, представляющий комплексные числа. Естественным образом комплексное число можно представить в памяти в виде следующей структуры C:

```
typedef struct Complex {
    double      x;
    double      y;
} Complex;
```

Нам нужно будет передавать этот тип по ссылке, так как он слишком велик, чтобы уместиться в одном значении `Datum`.

В качестве внешнего строкового представления типа мы выберем строку вида `(x,y)`.

Функции ввода и вывода обычно несложно написать, особенно функцию вывода. Но определяя внешнее строковое представление типа, помните, что в конце концов вам придётся реализовать законченный и надёжный метод разбора этого представления в функции ввода. Например, так:

```
PG_FUNCTION_INFO_V1(complex_in);
```

```
Datum
complex_in(PG_FUNCTION_ARGS)
{
    char       *str = PG_GETARG_CSTRING(0);
    double      x,
                y;
    Complex    *result;

    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for complex: \"%s\"",
                        str)));

    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}
```

Функция вывода может быть простой:

```
PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    char       *result;

    result = psprintf("(%g,%g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);
}
```

Вам нужно позаботиться о том, чтобы функции ввода и вывода были обратными друг к другу. В противном случае вы столкнётесь с проблемами, когда вам потребуется выгрузить данные в файл, а затем прочитать их. Это особенно распространённая проблема, когда дело касается чисел с плавающей точкой.

Дополнительно пользовательский тип может предоставлять функции для ввода и вывода в двоичном виде. Двоичный ввод/вывод обычно работает быстрее, но хуже портируется, чем текстовый. Как и с текстовым представлением, выбор, каким будет двоичное представление, остаётся за вами. Многие встроенные типы данных стараются обеспечить двоичное представление, независимое от машинной архитектуры. Для типа `complex` мы воспользуемся функциями двоичного ввода/вывода типа `float8`:

```
PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
{
    StringInfo  buf = (StringInfo) PG_GETARG_POINTER(0);
    Complex    *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
```

```
    PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(complex_send);

Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;

    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

Написав функции ввода/вывода и скомпилировав их в разделяемую библиотеку, мы можем определить тип `complex` в SQL. Сначала мы объявим его как тип-пустышку:

```
CREATE TYPE complex;
```

Это позволит нам ссылаться на этот тип, определяя для него функции ввода/вывода. Теперь мы определим функции ввода/вывода:

```
CREATE FUNCTION complex_in(cstring)
    RETURNS complex
    AS 'имя_файла'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
    RETURNS cstring
    AS 'имя_файла'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_recv(internal)
   RETURNS complex
   AS 'имя_файла'
   LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_send(complex)
   RETURNS bytea
   AS 'имя_файла'
   LANGUAGE C IMMUTABLE STRICT;
```

Наконец, мы можем предоставить полное определение типа данных:

```
CREATE TYPE complex (
   internallength = 16,
   input = complex_in,
   output = complex_out,
   receive = complex_recv,
   send = complex_send,
   alignment = double
);
```

Когда определяется новый базовый тип, Postgres Pro автоматически обеспечивает поддержку массивов с элементами такого типа. Тип массива обычно получает имя по имени базового типа с добавленным спереди символом подчёркивания (_).

Когда тип данных определён, мы можем объявить дополнительные функции для выполнения полезных операций с этим типом. Затем поверх этих функций могут быть определены операторы, а если потребуется, и классы операторов, для поддержки индексации этого типа. Эти дополнительные уровни обсуждаются в следующих разделах.

Если внутреннее представление типа данных имеет переменную длину, оно должно соответствовать стандартной схеме данных переменной длины: первые четыре байта должно занимать поле `char[4]`, к которому никогда не следует обращаться напрямую (по обыкновению названное `vl_len_`). Чтобы сохранить в этом поле размер элемента (включая длину самого поля), вы должны использовать макрос `SET_VARSIZE()`, а чтобы получить его — макрос `VARSIZE()`. (Эти макросы нужны, потому что поле длины может кодироваться по-разному на разных платформах.)

За дополнительными подробностями обратитесь к команде CREATE TYPE.

35.11.1. Особенности TOAST

Если значения вашего типа данных могут быть разного размера (во внутренней форме), обычно для такого типа желательно реализовать поддержку TOAST (см. Раздел 63.2). Это следует делать, даже если значения слишком малы для сжатия или внешнего хранения, так как TOAST позволяет сэкономить пространство и с данными маленького размера, сокращая издержки в заголовке.

Для поддержки хранения TOAST функции на C, работающие с таким типом данных, должны позаботиться о распаковке поступивших им данных, используя макрос `PG_DETOAST_DATUM`. (Эту внутреннюю особенность обычно скрывает дополнительный, определяемый для типа макрос `GETARG_DATATYPE_P`.) Затем, выполняя команду `CREATE TYPE`, укажите в качестве внутренней длины `variable` и выберите подходящий вариант хранения (не `plain`).

Если выравнивание данных не имеет значения (либо только для некоторой функции, либо потому что для типа данных в любом случае применяется выравнивание по байтам), некоторых издержек, связанных с макросом `PG_DETOAST_DATUM`, можно избежать. Вместо него можно использовать `PG_DETOAST_DATUM_PACKED` (его обычно скрывает определяемый для типа макрос `GETARG_DATATYPE_PP`) и воспользоваться макросами `VARSIZE_ANY_EXHDR` и `VARDATA_ANY` для обращения к потенциально сжатым данным. Стоит ещё раз отметить, что данные, возвращаемые этими макросами, не выравниваются, даже если выравнивание задано в определении типа. Если выравнивание важно, вы должны задействовать обычный интерфейс `PG_DETOAST_DATUM`.

> **Примечание**
>
> В старом коде поле `vl_len_` часто объявлялось как `int32`, а не `char[4]`. Это ничем не чревато до той поры, пока в определении структуры имеются другие поля с выравниванием как минимум `int32`. Но с потенциально невыровненными данными такое определение структуры опасно; компилятор может воспринять его как право полагать, что данные выровнены, что может привести к аварийным сбоям в архитектурах, строгих к выравниванию.

Поддержка TOAST даёт также возможность иметь *развёрнутое* представление данных в памяти, работать с которым будет удобнее, чем с форматом хранения на диске. Обычный или «плоский» формат хранения varlena в конце концов представляет собой просто набор байт; он не может содержать указатели, так как эти байты могут быть скопированы в другие адреса. Для сложных типов данных работать с плоским форматом данных может быть довольно дорого, так что Postgres Pro даёт возможность «развернуть» плоский формат в представление, более подходящее для вычислений, и затем передавать эту структуру в памяти функциям, работающим с этим типом.

Для использования развёрнутого хранения тип данных должен определять развёрнутый формат по правилам, описанным в `src/include/utils/expandeddatum.h`, и предоставлять функции для «разворачивания» плоского значения в этот формат, а также для «заворачивания» этого формата опять в обычное представление varlena. Затем надо добиться, чтобы все функции на C могли принимать любое представление, возможно выполняя преобразование одного в другое

непосредственно при получении. Для этого не требуется исправлять сразу все существующие функции для этого типа данных, так как имеющийся стандартный макрос `PG_DETOAST_DATUM` способен преобразовывать развёрнутые входные данные в обычный плоский формат. Таким образом, все существующие функции, работающие с плоским форматом varlena продолжат работать, хотя и не очень эффективно, с развёрнутыми входными данными; их необязательно переделывать, пока не потребуется оптимизировать производительность.

Функции на C, умеющие работать с развёрнутым представлением, обычно делятся на две категории: те, что могут работать с развёрнутым форматом, и те, что могут принимать и развёрнутые, и плоские данные varlena. Первые проще написать, но они могут быть менее эффективными в целом, так как преобразование плоского значения в развёрнутую форму для использования только одной функцией может стоить больше, чем сэкономится при обработке данных в развёрнутом формате. Когда нужно работать только с развёрнутым форматом, преобразование плоских значений в развёрнутую форму можно скрыть в макросе, извлекающем аргументы, чтобы функция была не сложнее, чем работающая с традиционными входными данными varlena. Чтобы принимать оба варианта входных значений, напишите функцию извлечения аргументов, которая будет распаковывать значения с сокращённым заголовком, а также внешние и сжатые, но не развёрнутые данные. Такую функцию можно определить как возвращающую указатель на объединение плоского формата varlena и развёрнутого формата. Какой формат получен фактически, вызывающий код может определить, вызвав макрос `VARATT_IS_EXPANDED_HEADER()`.

Инфраструктура TOAST позволяет не только отличить обычные значения varlena от развёрнутых значений, но и различить указатели «для чтения/записи» и «только для чтения» на развёрнутые значения. Функции на C, которым нужно читать развёрнутое значение, или которые будут менять его безопасным и невидимым извне образом, могут не обращать внимания на тип полученного указателя. Если же функции на C выдают изменённую версию входного значения, они могут изменять развёрнутые входные данные на месте, только когда получают указатель для чтения/записи, но не когда получен указатель только для чтения. В последнем случае они должны сначала скопировать значение и получить новое значение, допускающее изменение. Функция на C, создающая новое развёрнутое значение, должна всегда возвращать указатель на него для чтения/записи. Кроме того, функция, изменяющая развёрнутое значение непосредственно по указателю для чтения/записи должна позаботиться о том, чтобы это значение осталось в приемлемом состоянии, если она отработает не полностью.

Примеры работы с развёрнутыми значениями можно найти в стандартной инфраструктуре массивов, в частности в `src/backend/utils/adt/array_expanded.c`.

35.12. Пользовательские операторы

Любой оператор представляет собой «синтаксический сахар» для вызова нижележащей функции, выполняющей реальную работу; поэтому прежде чем вы сможете создать оператор, необходимо создать нижележащую функцию. Однако оператор — *не исключительно* синтаксический сахар, так как он несёт и дополнительную информацию, помогающую планировщику запросов оптимизировать запросы с этим оператором. Рассмотрению этой дополнительной информации будет посвящён следующий раздел.

Postgres Pro поддерживает левые унарные, правые унарные и бинарные операторы. Операторы могут быть перегружены; то есть одно имя оператора могут иметь различные операторы с разным количеством и типами операндов. Когда выполняется запрос, система определяет, какой оператор вызвать, по количеству и типам предоставленных операндов.

В следующем примере создаётся оператор сложения двух комплексных чисел. Предполагается, что мы уже создали определение типа `complex` (см. Раздел 35.11). Сначала нам нужна функция, собственно выполняющая операцию, а затем мы сможем определить оператор:

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'имя_файла', 'complex_add'
```

```
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_add,
    commutator = +
);
```

Теперь мы можем выполнить такой запрос:

```
SELECT (a + b) AS c FROM test_complex;

        c
-----------------
 (5.2,6.05)
 (133.42,144.95)
```

Мы продемонстрировали создание бинарного оператора. Чтобы создать унарный оператор, просто опустите `leftarg` (для левого унарного) или `rightarg` (для правого унарного). Обязательными в `CREATE OPERATOR` являются только предложение `procedure` и объявления аргументов. Предложение `commutator`, добавленное в данном примере, представляет необязательную подсказку для оптимизатора запросов. Подробнее о `commutator` и других подсказках для оптимизатора рассказывается в следующем разделе.

35.13. Информация для оптимизации операторов

Определение оператора в Postgres Pro может включать различные дополнительные предложения, которые сообщают системе полезные сведения о поведении оператора. Старайтесь задавать эти предложения при возможности, так как они могут значительно ускорить выполнение запросов, использующих данный оператор. Но если вы задаёте их, убедитесь, что они корректны! Неправильное применение предложений оптимизации может привести к замедлению запросов, неочевидным ошибочным результатам и другим неприятностям. Если вы не уверены в правильности предложения оптимизации, лучше вовсе не использовать его; единственным последствием будет то, что запросы будут работать медленнее, чем могли бы.

В будущих версиях Postgres Pro могут быть добавлены и другие предложения. Здесь описываются те, что поддерживаются версией 10.23.1.

35.13.1. `COMMUTATOR`

Предложение `COMMUTATOR`, если представлено, задаёт оператор, коммутирующий для определяемого. Оператор A является коммутирующим для оператора B, если (x A y) равняется (y B x) для всех возможных значений x, y. Заметьте, что B также будет коммутирующим для A. Например, операторы `<` и `>` для конкретного типа данных обычно являются коммутирующими друг для друга, а оператор `+` — коммутирующий для себя. Но традиционный оператор `-` коммутирующего не имеет.

Тип левого операнда оператора должен совпадать с типом правого операнда коммутирующего для него оператора, и наоборот. Поэтому имя коммутирующего оператора — это всё, что Postgres Pro должен знать, чтобы найти коммутатор, и всё, что нужно указать в предложении `COMMUTATOR`.

Информация о коммутирующих операторах крайне важна для операторов, которые будут применяться в индексах и условиях соединения, так как, используя её, оптимизатор запросов может «переворачивать» такие выражения и получать формы, необходимые для разных типов планов. Например, рассмотрим запрос с предложением WHERE `tab1.x = tab2.y`, где `tab1.x` и `tab2.y` имеют пользовательский тип, и предположим, что у нас есть индекс по столбцу `tab2.y`. Оптимизатор сможет задействовать сканирование по индексу, только если ему удастся перевернуть выражение `tab2.y = tab1.x`, так как механизм сканирования по индексу ожидает, что

индексируемый столбец находится слева от оператора. Postgres Pro сам по себе *не* будет полагать, что такое преобразование возможно — это должен определить создатель оператора =, добавив информацию о коммутирующем операторе.

Когда вы определяете оператор, коммутирующий сам для себя, вы делаете именно это. Если же вы определяете пару коммутирующих операторов, возникает небольшое затруднение: как оператор, определяемый первым, может ссылаться на другой, ещё не определённый? Есть два решения этой проблемы:

- Во-первых, можно опустить предложение COMMUTATOR для первого оператора, который вы определяете, а затем добавить его в определении второго. Так как Postgres Pro знает, что коммутирующие операторы связаны парами, встречая второе определение, он автоматически возвращается к первому и добавляет в него недостающее предложение COMMUTATOR.
- Во-вторых, можно добавить предложение COMMUTATOR в оба определения. Когда Postgres Pro обрабатывает первое определение и видит, что COMMUTATOR ссылается на несуществующий оператор, в системном каталоге создаётся фиктивная запись для этого оператора. В этой фиктивной записи актуальны будут только имя оператора, типы левого и правого операндов, а также тип результата, так как это всё, что Postgres Pro может определить в этот момент. Запись первого оператора будет связана с этой фиктивной записью. Затем, когда вы определите второй оператор, система внесёт в эту фиктивную запись дополнительную информацию из второго определения. Если вы попытаетесь применить фиктивный оператор, прежде чем он будет полностью определён, вы просто получите сообщение об ошибке.

35.13.2. NEGATOR

Предложение NEGATOR, если присутствует, задаёт оператор, обратный к определяемому. Оператор A является обратным к оператору B, если они оба возвращают логический результат и (x A y) равняется NOT (x B y) для всех возможных x, y. Заметьте, что B так же является обратным к A. Например, операторы < и >= составляют пару обратных друг к другу для большинства типов данных. Никакой оператор не может быть обратным к себе же.

В отличие от коммутирующих операторов, два унарных оператора вполне могут быть обратными к друг другу; это будет означать, что (A x) равняется NOT (B x) для всех x (и для правых унарных операторов аналогично).

У оператора, обратного данному, типы левого и/или правого операнда должны соответствовать типам данного оператора, так же как и с предложением COMMUTATOR; отличие только в том, что имя оператора задаётся в предложении NEGATOR.

Указание обратного оператора очень полезно для оптимизатора запросов, так как это позволяет упростить выражение вида `NOT (x = y)` до `x <> y`. Такие выражения не так редки, как может показаться, так как операции NOT могут добавляться автоматически в результате реорганизаций выражений.

Пару обратных операторов можно определить теми же способами, что были описаны ранее для пары коммутирующих.

35.13.3. RESTRICT

Предложение RESTRICT, если представлено, определяет функцию оценки избирательности ограничения для оператора. (Заметьте, что в нём задаётся имя функции, а не оператора.) Предложения RESTRICT имеют смысл только для бинарных операторов, возвращающих boolean. Идея оценки избирательности ограничения заключается в том, чтобы определить, какой процент строк в таблице будет удовлетворять условию WHERE вида:

```
column OP constant
```

для текущего оператора и определённого значения константы. Это помогает оптимизатору примерно определить, сколько строк будет исключено предложениями WHERE такого вида. (ВЫ

спросите, а что если константа находится слева? Ну, собственно для таких случаев и задаётся COMMUTATOR...)

Рамки данной главы не позволяют описать разработку новых функций оценки избирательности ограничения, но обычно можно использовать один из стандартных системных оценщиков для большинства дополнительных операторов. Стандартные оценщики ограничений следующие:

eqsel для =
neqsel для <>
scalarltsel для < или <=
scalargtsel для > или >=

Может показаться немного странным, что выбраны именно эти категории, но, если подумать, это имеет смысл. Оператор = обычно оставляет только небольшой процент строк в таблице, а <> отбрасывает то же количество. Оператор < оставляет процент, зависящий от того, в какой диапазон значений определённого столбца таблицы попадает заданная константа (информация об этих диапазонах собирается командой ANALYZE и предоставляется оценщику избирательности). Оператор <= оставляет чуть больший процент, чем <, при сравнении с той же константой, но они настолько близки, что различать их не имеет смысла, так как это не даст лучшего результата, чем просто угадывание. Подобные замечания применимы и к операторам > и >=.

Часто вы можете обойтись функциями eqsel и neqsel для операторов с очень высокой или низкой избирательностью, даже если это не операторы собственно равенства или неравенства. Например, геометрические операторы приблизительного равенства используют eqsel в предположении, что соответствующие (равные) элементы будут составлять только небольшой процент от всех записей таблицы.

Функции scalarltsel и scalargtsel можно использовать для сравнений с типами данных, которые могут быть каким-либо осмысленным образом преобразованы в числовые скалярные значения для сравнения диапазонов. Если возможно, добавьте свой тип данных в число типов, которые понимает функция convert_to_scalar(), реализованная в src/backend/utils/adt/selfuncs.c. (Когда-нибудь эта функция должна быть заменена специализированными функциями, которые будут устанавливаться для конкретных типов в определённом столбце системного каталога pg_type; но сейчас это не так.) Если вы этого не сделаете, всё будет работать, но оценки оптимизатора будут не так хороши, как могли бы быть.

Для геометрических операторов разработаны дополнительные функции оценки избирательности в src/backend/utils/adt/geo_selfuncs.c: areasel, positionsel и contsel. На момент написания документации это просто заглушки, но вы тем не менее вполне можете использовать (или, ещё лучше, доработать) их.

35.13.4. JOIN

Предложение JOIN, если представлено, определяет функцию оценки избирательности соединения для оператора. (Заметьте, что в нём задаётся имя функции, а не оператора.) Предложения JOIN имеют смысл только для бинарных операторов, возвращающих boolean. Идея оценки избирательности соединения заключается в том, чтобы угадать, какой процент строк в паре таблиц будет удовлетворять условию WHERE следующего вида:

```
table1.column1 OP table2.column2
```

для текущего оператора. Как и RESTRICT, это предложение очень помогает оптимизатору, позволяя ему выяснить, какой из возможных вариантов соединения скорее всего окажется выгоднее.

Как и ранее, в этой главе мы не будем пытаться рассказать, как написать функцию оценивания избирательности соединения, а просто отметим, что вы можете использовать один из подходящих стандартных оценщиков:

eqjoinsel для =
neqjoinsel для <>
scalarltjoinsel для < или <=

`scalargtjoinsel` для > или >=
`areajoinsel` для сравнений областей в плоскости
`positionjoinsel` для сравнения положений в плоскости
`contjoinsel` для проверки на включение в плоскости

35.13.5. HASHES

Предложение `HASHES`, если присутствует, говорит системе, что для соединений с применением этого оператора допустимо использовать метод соединения по хешу. `HASHES` имеет смысл только для бинарного оператора, который возвращает `boolean`, и на практике этот оператор должен выражать равенство значений некоторого типа данных или пары типов данных.

Соединение по хешу базируется на том предположении, что оператор соединения возвращает истину только для таких пар значений слева и справа, для которых получается одинаковый хеш. Если два значения оказываются в разных ячейках хеша, операция соединения никогда не будет сравнивать их, неявно подразумевая, что результат оператора соединения в этом случае должен быть ложным. Поэтому не имеет никакого смысла указывать `HASHES` для операторов, которые не представляют какую-любо форму равенства. В большинстве случаев практический смысл в поддержке хеширования есть только для операторов, принимающих один тип данных с обеих сторон. Однако иногда возможно разработать хеш-функции, совместимые сразу с несколькими типами данных; то есть, функции, которые будут выдавать одинаковые хеш-коды для «равных» значений, несмотря на то, что эти значения будут представлены по-разному. Например, довольно легко функции с такой особенностью реализуются для хеширования целых чисел различного размера.

Чтобы оператор соединения имел характеристику `HASHES`, он должен входить в семейство операторов индексирования по хешу. Это требование откладывается, когда оператор только создаётся, ведь нужное семейство операторов, разумеется, ещё не может существовать. Но при попытке использовать такой оператор для соединения по хешу, возникнет ошибка во время выполнения, если такого семейства не окажется. Системе необходимо знать семейство операторов, чтобы найти функции для хеширования типа(ов) входных данных оператора. Конечно, вы должны также определить подходящие функции хеширования, прежде чем сможете создать семейство операторов.

При подготовке функции хеширования обязательно позаботьтесь о том, чтобы она всегда выдавала нужный результат, вне зависимости от особенностей машинной архитектуры. Например, если ваш тип данных представлен в структуре, в которой есть незначащие дополняющие биты, нельзя просто передать всю структуру функции `hash_any`. (Это возможно, только если все ваши операторы и функции гарантированно очищают незначащие биты, что является рекомендуемой стратегией.) В качестве другого примера можно привести типы с плавающей точкой в стандарте IEEE, в которых отрицательный ноль и положительный ноль — различные значения (отличаются на уровне битов), но при сравнении они считаются равными. Если значение с плавающей точкой может содержать отрицательный ноль, требуются дополнительные действия, чтобы для него выдавался тот же хеш, что и для положительного нуля.

Оператор соединения по хешу должен иметь коммутирующий (это может быть тот же оператор, если у него два операнда одного типа, либо связанный оператор равенства, в противном случае), относящийся к тому же семейству операторов. В случае его отсутствия, при попытке использования оператора возможны ошибки планировщика. Также желательно (хотя это строго не требуется), чтобы в семействе операторов хеширования, поддерживающем несколько типов данных, определялись операторы равенства для всех комбинаций этих типов данных; это способствует лучшей оптимизации.

> **Примечание**
>
> Функция, реализующая оператор соединения по хешу, должна быть постоянной (IMMUTABLE) или стабильной (STABLE). Если эта функция изменчивая (VOLATILE), система никогда не будет применять этот оператор для соединения по хешу.

> **Примечание**
>
> Если оператор соединения по хешу реализуется строгой функцией (STRICT), эта функция также должна быть полной: то есть она должна возвращать true или false, но не NULL, для любых двух аргументов, отличных от NULL. Если это правило не соблюдается, оптимизация операций `IN` с хешем может приводить к неверным результатам. (В частности, выражение `IN` может вернуть false, когда правильным ответом, согласно стандарту, должен быть NULL, либо выдать ошибку с сообщением о том, что оно не готов к результату NULL.)

35.13.6. `MERGES`

Предложение `MERGES`, если присутствует, говорит системе, что для соединений с применением этого оператора допустимо использовать метод соединения слиянием. `MERGES` имеет смысл только для бинарного оператора, который возвращает `boolean`, и на практике этот оператор должен выражать равенство значений некоторого типа данных или пары типов данных.

Идея объединения слиянием заключается в упорядочивании таблиц слева и справа и затем параллельном сканировании их. Поэтому оба типа данных должны поддерживать сортировку в полном объёме, а оператор соединения должен давать положительный результат только для пар значений, оказавшихся в «одном месте» при определённом порядке сортировки. На практике это означает, что оператор соединения должен работать как проверка на равенство. Но при этом возможно объединить слиянием два различных типа данных, если они совместимы логически. Например, оператор проверки равенства `smallint` и `integer` может применяться для соединений слиянием; понадобятся только операторы сортировки, приводящие оба типа данных в логически совместимые последовательности.

Чтобы оператор соединения имел характеристику `MERGES`, он должен являться членом семейства операторов индекса `btree`, реализующим равенство. Это требование откладывается, когда оператор только создаётся, ведь нужное семейство операторов, разумеется, ещё не может существовать. Но этот оператор не будет фактически применяться для соединений слиянием, пока не будет найдено соответствующее семейство операторов. Таким образом, флаг `MERGES` только подсказывает планировщику, что стоит обратиться к соответствующему семейству.

Оператор соединения слиянием должен иметь коммутирующий (это может быть тот же оператор, если у него два операнда одного типа, либо связанный оператор равенства, в противном случае), относящийся к тому же семейству операторов. В случае его отсутствия, при попытке использования оператора возможны ошибки планировщика. Также желательно (хотя это строго не требуется), чтобы в семействе операторов `btree`, поддерживающем несколько типов данных, определялись операторы равенства для всех комбинаций этих типов данных; это способствует лучшей оптимизации.

> **Примечание**
>
> Функция, реализующая оператор соединения слиянием, должна быть постоянной (IMMUTABLE) или стабильной (STABLE). Если эта функция изменчивая (VOLATILE), система никогда не будет применять этот оператор для соединения слиянием.

35.14. Интерфейсы расширений для индексов

Описанные до этого процедуры позволяли определять новые типы, функции и операторы. Однако мы ещё не можем определить индекс по столбцу нового типа данных. Для этого нам потребуется создать *класс операторов* для нового типа данных. Далее в этом разделе мы продемонстрируем эту концепцию на примере: мы создадим новый класс операторов для метода индекса-B-дерева,

в котором будут храниться комплексные числа и сортироваться по возрастанию абсолютного значения.

Классы операторов могут объединяться в *семейства операторов*, выражающие зависимости между семантически совместимыми классами. Когда вводится один тип данных, достаточно класса операторов, так что мы начнём с него, а к семействам операторов вернёмся позже.

35.14.1. Методы индексов и классы операторов

В системном каталоге есть таблица `pg_am`, содержащая записи для каждого метода индекса (внутри называемого методом доступа). Поддержка обычного доступа к таблицам встроена в Postgres Pro, но все методы доступа описываются в `pg_am`. Система позволяет добавлять новые методы доступа — для этого нужно написать необходимый код, а затем добавить запись в `pg_am`, но это выходит за рамки данной главы (см. Главу 56).

Процедуры метода индекса непосредственно ничего не знают о типах данных, с которыми будет применяться этот метод. Вместо этого, набор операций, которые нужны методу индекса для работы с конкретным типом данных, определяется *классом операторов*. Классы операторов называются так потому, что они определяют множество операторов в предложении `WHERE`, которые могут использоваться с индексом (т. е. могут быть сведены к сканированию индекса). В классе операторов могут также определяться некоторые *опорные процедуры*, необходимые для внутренних операций метода индекса, но они не соответствуют напрямую каким-либо операторам предложения `WHERE`, которые могут обрабатываться с индексом.

Для одного типа данных и метода индекса можно определить несколько классов операторов. Благодаря этому, для одного типа данных можно использовать несколько семантически разных вариантов индексирования. Например, индекс-B-дерево требует, чтобы для каждого типа данных, с которым он работает, определялся порядок сортировки. Для типа комплексных чисел может быть полезен класс операторов B-дерева, сортирующий данные по модулю комплексного числа, и ещё один, сортирующий по вещественной части, и т. п. Обычно предполагается, что один из классов операторов будет применяться чаще других, и тогда он помечается как класс по умолчанию для данного типа и метода индекса.

Одно и то же имя класса операторов может использоваться для разных методов индекса (например, для методов индекса-B-дерева или хеш-индекса применяются классы операторов `int4_ops`), но все такие классы являются независимыми и должны определяться отдельно.

35.14.2. Стратегии методов индексов

Операторам, которые связываются с классом операторов, назначаются «номера стратегий», определяющие роль каждого оператора в контексте его класса. Например, в B-дереве должен быть строгий порядок ключей с отношениями меньше/больше, так что в данном контексте представляют интерес операторы «меньше» и «больше или равно». Так как Postgres Pro позволяет пользователям определять операторы произвольным образом, Postgres Pro не может просто посмотреть на имя оператора (`<` или `>=`) и сказать, какое сравнение он выполняет. Вместо этого для метода индекса определяется набор «стратегий», которые можно считать обобщёнными операторами. Каждый класс операторов устанавливает, какие фактические операторы соответствуют стратегиям для определённого типа данных и интерпретации семантики индекса.

Для метода индекса-B-дерева определены пять стратегий, описанных в Таблице 35.2.

Таблица 35.2. Стратегии B-дерева

Операция	Номер стратегии
меньше	1
меньше или равно	2
равно	3

Операция	Номер стратегии
больше или равно	4
больше	5

Индексы по хешу поддерживают только сравнение на равенство, так что они используют только одну стратегию, показанную в Таблице 35.3.

Таблица 35.3. Стратегии хеша

Операция	Номер стратегии
равно	1

Индексы GiST более гибкие: для них вообще нет фиксированного набора стратегий. Вместо этого опорная процедура «согласованности» каждого конкретного класса операторов GiST интерпретирует номера стратегий как ей угодно. Например, некоторые из встроенных классов операторов для индексов GiST индексируют двумерные геометрические объекты, и реализуют стратегии «R-дерева», показанные в Таблице 35.4. Четыре из них являются истинно двумерными проверками (overlaps, same, contains, contained by); другие четыре учитывают только ординаты, а ещё четыре проводят же проверки только с абсциссами.

Таблица 35.4. Стратегии двумерного «R-дерева» индекса GiST

Операция	Номер стратегии
строго слева от	1
не простирается правее	2
пересекается с	3
не простирается левее	4
строго справа от	5
одинаковы	6
содержит	7
содержится в	8
не простирается выше	9
строго ниже	10
строго выше	11
не простирается ниже	12

Индексы SP-GiST такие же гибкие, как и индексы GiST: для них не задаётся фиксированный набор стратегий. Вместо этого опорные процедуры каждого класса операторов интерпретируют номера стратегий в соответствии с определением класса операторов. В качестве примера, в Таблице 35.5 приведены номера стратегий, установленные для встроенных классов операторов для точек.

Таблица 35.5. Стратегии SP-GiST для точек

Операция	Номер стратегии
строго слева от	1
строго справа от	5
одинаковы	6
содержится в	8
строго ниже	10

Операция	Номер стратегии
строго выше	11

Индексы GIN такие же гибкие, как и индексы GiST и SP-GiST: для них не задаётся фиксированный набор стратегий. Вместо этого опорные процедуры каждого класса операторов интерпретируют номера стратегий в соответствии с определением класса операторов. В качестве примера, в Таблице 35.6 приведены номера стратегий, установленные для встроенного класса операторов для массивов.

Таблица 35.6. Стратегии GIN для массивов

Операция	Номер стратегии
пересекается с	1
содержит	2
содержится в	3
равно	4

Индексы BRIN такие же гибкие, как и индексы GiST, SP-GiST и GIN: для них не задаётся фиксированный набор стратегий. Вместо этого опорные процедуры каждого класса операторов интерпретируют номера стратегий в соответствии с определением класса операторов. В качестве примера, в Таблице 35.7 приведены номера стратегий, используемые встроенными классами операторов `Minmax`.

Таблица 35.7. Стратегии BRIN Minmax

Операция	Номер стратегии
меньше	1
меньше или равно	2
равно	3
больше или равно	4
больше	5

Заметьте, что все вышеперечисленные операторы возвращают булевы значения. На практике все операторы, определённые как операторы поиска для метода индекса, должны возвращать тип `boolean`, так как они должны находиться на верхнем уровне предложения `WHERE`, чтобы для них применялся индекс. (Некоторые методы доступа по индексу также поддерживают *операторы упорядочивания*, которые обычно не возвращают булевы значения; это обсуждается в Подразделе 35.14.7.)

35.14.3. Опорные процедуры метода индекса

Стратегии обычно не дают системе достаточно информации, чтобы понять, как использовать индекс. На практике, чтобы методы индекса работали, необходимы дополнительные опорные процедуры. Например, метод индекса-B-дерева должен уметь сравнивать два ключа и определять, больше, равен или меньше ли первый второго. Аналогично, метод индекса по хешу должен уметь сравнивать хеш-коды значений ключа. Эти операции не соответствуют операторам, которые применяются в условиях в командах SQL; это внутрисистемные подпрограммы, используемые методами индекса.

Так же, как и со стратегиями, класс операторов определяет, какие конкретные функции должны играть каждую из ролей для определённого типа данных и интерпретации семантики индекса. Для метода индекса определяется набор нужных ему функций, а класс оператора выбирает нужные функции для применения, назначая им «номера опорных функций», определяемые методом индекса.

Для B-деревьев требуется одна опорная функция, а вторая задаётся по выбору разработчика класса операторов, как показано в Таблице 35.8.

Таблица 35.8. Опорные функции B-деревьев

Функция	Номер опорной функции
Сравнивает два ключа и возвращает целое меньше нуля, ноль или целое больше нуля, показывающее, что первый ключ меньше, равен или больше второго	1
Возвращает адрес вызываемой из C функции(й), поддерживающей сортировку, как описано в `utils/sortsupport.h` (необязательная)	2

Для хеш-индексов требуется одна опорная функция, указанная в Таблице 35.9.

Таблица 35.9. Опорные функции хеша

Функция	Номер опорной функции
Вычисляет значение хеша для ключа	1

Для индексов GiST предусмотрены девять опорных функций, две из которых необязательные; они описаны в Таблице 35.10. (За дополнительными сведениями обратитесь к Главе 58.)

Таблица 35.10. Опорные функции GiST

Функция	Описание	Номер опорной функции
`consistent`	определяет, удовлетворяет ли ключ условию запроса	1
`union`	вычисляет объединение набора ключей	2
`compress`	вычисляет сжатое представление ключа или индексируемого значения	3
`decompress`	вычисляет развёрнутое представление сжатого ключа	4
`penalty`	вычисляет стоимость добавления нового ключа в поддерево с заданным ключом	5
`picksplit`	определяет, какие записи страницы должны быть перемещены в новую страницу, и вычисляет ключи объединения для результирующих страниц	6
`equal`	сравнивает два ключа и возвращает true, если они равны	7
`distance`	определяет дистанцию от ключа до искомого значения (необязательная)	8
`fetch`	вычисляет исходное представление сжатого ключа для сканирования только по индексу (необязательная)	9

Для индексов SP-GiST требуются пять опорных функций, описанных в Таблице 35.11. (За дополнительными сведениями обратитесь к Главе 59.)

Таблица 35.11. Опорные функции SP-GiST

Функция	Описание	Номер опорной функции
`config`	предоставляет основную информацию о классе операторов	1
`choose`	определяет, как вставить новое значение во внутренний элемент	2
`picksplit`	определяет, как разделить множество значений	3
`inner_consistent`	определяет, в каких внутренних ветвях нужно искать заданное значение	4
`leaf_consistent`	определяет, удовлетворяет ли ключ условию запроса	5

Для индексов GIN предусмотрены шесть опорных функций, три из которых необязательные; они описаны в Таблице 35.12. (За дополнительными сведениями обратитесь к Главе 60.)

Таблица 35.12. Опорные функции GIN

Функция	Описание	Номер опорной функции
`compare`	сравнивает два ключа и возвращает целое меньше нуля, ноль или целое больше нуля, показывающее, что первый ключ меньше, равен или больше второго	1
`extractValue`	извлекает ключи из индексируемого значения	2
`extractQuery`	извлекает ключи из условия запроса	3
`consistent`	определяет, соответствует ли значение условию запроса (логическая вариация) (не требуется, если присутствует опорная функция 6)	4
`comparePartial`	сравнивает частичный ключ из запроса с ключом из индекса и возвращает целое число меньше нуля, ноль или больше нуля, показывающее, что GIN должен игнорировать эту запись индекса, принять её как соответствующую или прекратить сканирование индекса (необязательная)	5
`triConsistent`	определяет, соответствует ли значение условию запроса (троичная вариация) (не	6

Функция	Описание	Номер опорной функции
	требуется, если присутствует опорная функция 4)	

Для индексов BRIN предусмотрены четыре базовые опорные функции, перечисленные в Таблице 35.13; для этих базовых функций может потребоваться предоставить дополнительные опорные функции. (За дополнительными сведениями обратитесь к Разделу 61.3.)

Таблица 35.13. Опорные функции BRIN

Функция	Описание	Номер опорной функции
`opcInfo`	возвращает внутреннюю информацию, описывающую сводные данные по индексированным столбцам	1
`add_value`	добавляет новое значение в существующий сводный кортеж индекса	2
`consistent`	определяет, удовлетворяет ли значение условию запроса	3
`union`	вычисляет объединение двух обобщающих кортежей	4

В отличие от операторов поиска, опорные функции возвращают тот тип данных, который ожидает конкретный метод индекса; например, функция сравнения для B-деревьев возвращает знаковое целое. Количество и типы аргументов для каждой опорной функции так же зависят от метода индекса. Для методов B-дерева и хеша функции сравнения и хеширования принимают те же типы данных, что и операторы, включённые в класс операторов, но для большинства опорных функций GiST, SP-GiST, GIN и BRIN это не так.

35.14.4. Пример

Теперь, когда мы познакомились с основными идеями, мы можем перейти к обещанному примеру создания нового класса операторов. (Рабочую копию этого примера можно найти в `src/tutorial/complex.c` и `src/tutorial/complex.sql` в пакете исходного кода.) Класс операторов включает операторы, сортирующие комплексные числа по порядку абсолютных значений, поэтому мы выбрали для него имя `complex_abs_ops`. Во-первых, нам понадобится набор операторов. Процедура определения операторов была рассмотрена в Разделе 35.12. Для класса операторов B-деревьев нам понадобятся операторы:

- абсолютное-значение меньше (стратегия 1)
- абсолютное-значение меньше-или-равно (стратегия 2)
- абсолютное-значение равно (стратегия 3)
- абсолютное-значение больше-или-равно (стратегия 4)
- абсолютное-значение больше (стратегия 5)

Чтобы не провоцировать ошибки при определении связанного набора операторов сравнения, лучше всего сначала написать вспомогательную функцию сравнения для B-дерева, а затем написать другие функции как однострочные оболочки этой вспомогательной функции. Это уменьшит вероятность получения несогласованных результатов в исключительных случаях. Следуя этому подходу, мы сначала напишем:

```
#define Mag(c)  ((c)->x*(c)->x + (c)->y*(c)->y)

static int
complex_abs_cmp_internal(Complex *a, Complex *b)
{
    double      amag = Mag(a),
```

```
                bmag = Mag(b);

    if (amag < bmag)
        return -1;
    if (amag > bmag)
        return 1;
    return 0;
}
```

Теперь функция «меньше» будет выглядеть так:

```
PG_FUNCTION_INFO_V1(complex_abs_lt);

Datum
complex_abs_lt(PG_FUNCTION_ARGS)
{
    Complex    *a = (Complex *) PG_GETARG_POINTER(0);
    Complex    *b = (Complex *) PG_GETARG_POINTER(1);

    PG_RETURN_BOOL(complex_abs_cmp_internal(a, b) < 0);
}
```

Остальные четыре функции отличаются от неё только тем, как сравнивают результат внутренней функции с нулём.

Затем мы объявим в SQL функции и операторы на основе этих функций:

```
CREATE FUNCTION complex_abs_lt(complex, complex) RETURNS bool
    AS 'имя_файла', 'complex_abs_lt'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR < (
   leftarg = complex, rightarg = complex, procedure = complex_abs_lt,
   commutator = > , negator = >= ,
   restrict = scalarltsel, join = scalarltjoinsel
);
```

Важно правильно определить обратные и коммутирующие операторы, а также подходящие функции избирательности ограничения и соединения; иначе оптимизатор не сможет использовать наш индекс эффективно. Заметьте, что для случаев меньше, равно и больше следует использовать другие функции оценки избирательности.

Здесь также стоит обратить внимание на следующее:

- Учтите, что может быть только один оператор с именем, например, =, который будет принимать тип `complex` с двух сторон. В этом случае у нас не будет другого оператора = для `complex`, но если мы создаём практически полезный тип данных, вероятно, мы захотим, чтобы оператор = проверял обычное равенство двух комплексных чисел (а не равенство их абсолютных значений). В этом случае для `complex_abs_eq` нужно выбрать какое-то другое имя оператора.
- Хотя в Postgres Pro разные функции могут иметь одинаковые имена SQL, если у них различные типы аргументов, в C только одна глобальная функция может иметь заданное имя. Поэтому не следует давать функции на C имя вроде `abs_eq`. Во избежание конфликтов с функциями для других типов данных, в имя функции на C обычно включается имя конкретного типа данных.
- Мы могли быть дать нашей функции имя `abs_eq` в SQL, рассчитывая на то, что Postgres Pro отличит её от любых других одноимённых функций SQL по типам аргументов. Но в данном случае для упрощения примера мы дали ей одинаковые имена на уровне C и уровне SQL.

На следующем этапе регистрируется опорная процедура, необходимая для B-деревьев. В нашем примере код C, реализующий её, находится в том же файле, что и функции операторов. Мы объявляем эту процедуру так:

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS 'имя_файла'
    LANGUAGE C IMMUTABLE STRICT;
```

Теперь, когда мы объявили требуемые операторы и опорную функцию, мы наконец можем создать класс операторов:

```
CREATE OPERATOR CLASS complex_abs_ops
    DEFAULT FOR TYPE complex USING btree AS
        OPERATOR        1       < ,
        OPERATOR        2       <= ,
        OPERATOR        3       = ,
        OPERATOR        4       >= ,
        OPERATOR        5       > ,
        FUNCTION        1       complex_abs_cmp(complex, complex);
```

Вот и всё! Теперь должно быть возможно создавать и использовать индексы-B-деревья по столбцам `complex`.

Операторы можно было записать более многословно, например, так:

```
        OPERATOR        1       < (complex, complex) ,
```

но в этом необходимости, так как эти операторы принимают тот же тип данных, для которого определяется класс операторов.

В приведённом примере предполагается, что этот класс операторов будет классом операторов B-дерева по умолчанию для типа `complex`. Если вам это не нужно, просто опустите слово `DEFAULT`.

35.14.5. Семейства и классы операторов

До этого мы неявно полагали, что класс операторов работает только с одним типом данных. Хотя в конкретном индексируемом столбце, определённо, может быть только один тип данных, часто бывает полезно индексировать операции, сравнивающие значение столбца со значением другого типа. Также, если в сочетании с классом операторов возможно применение оператора, работающего с двумя типами, для другого типа данных обычно тоже создаётся собственный класс. В таких случаях полезно установить явную связь между связанными классами, так как это поможет планировщику оптимизировать SQL-запросы (особенно для классов операторов B-дерева, потому что планировщик хорошо знает, как работать с ними).

Для удовлетворения этих потребностей в Postgres Pro введена концепция *семейства операторов*. Семейство операторов содержит один или несколько классов операторов и может также содержать индексируемые операторы и соответствующие опорные функции, принадлежащие к семейству в целом, но не к какому-то одному классу в нём. Мы называем такую связь операторов и функций с семейством «слабой», в отличие от обычной связи с определённым классом. Как правило, классы содержат операторы с операндами одного типа, тогда как межтиповые операторы слабо связываются с семейством.

Все операторы и функции в семействе операторов должны иметь совместимую семантику; требования к совместимости устанавливаются методом индекса. Вы можете спросить, зачем в таком случае вообще выделять конкретные подмножества семейства в виде классов операторов; и на самом деле во многих случаях деление на классы не имеет значения, важно только связывание с семейством. Смысл классов операторов в том, что они определяют, какая часть семейства необходима для поддержки некоторого индекса. Если существует индекс, использующий класс операторов, этот класс нельзя будет удалить, не удалив индекс — но другие части семейства, а

именно, другие классы операторов и слабосвязанные операторы, удалить можно. Таким образом, класс операторов должен определяться так, чтобы он содержал минимальный набор операторов и функций, обоснованно требующихся для работы с индексом по определённому типу данных, а несущественные операторы могут добавляться в качестве слабосвязанных членов в семейство операторов.

В качестве примера, в Postgres Pro есть встроенное семейство операторов B-дерева `integer_ops`, включающее классы операторов `int8_ops`, `int4_ops` и `int2_ops` для индексов по столбцам `bigint` (`int8`), `integer` (`int4`) и `smallint` (`int2`), соответственно. В этом семействе также содержатся операторы межтипового сравнения, позволяющие сравнивать значения любых двух этих типов, так что индексом по любому из этих типов можно пользоваться, выполняя сравнение с другим типом. Это семейство можно представить такими определениями:

```
CREATE OPERATOR FAMILY integer_ops USING btree;

CREATE OPERATOR CLASS int8_ops
DEFAULT FOR TYPE int8 USING btree FAMILY integer_ops AS
  -- standard int8 comparisons
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint8cmp(int8, int8) ,
  FUNCTION 2 btint8sortsupport(internal) ;

CREATE OPERATOR CLASS int4_ops
DEFAULT FOR TYPE int4 USING btree FAMILY integer_ops AS
  -- standard int4 comparisons
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint4cmp(int4, int4) ,
  FUNCTION 2 btint4sortsupport(internal) ;

CREATE OPERATOR CLASS int2_ops
DEFAULT FOR TYPE int2 USING btree FAMILY integer_ops AS
  -- standard int2 comparisons
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint2cmp(int2, int2) ,
  FUNCTION 2 btint2sortsupport(internal) ;

ALTER OPERATOR FAMILY integer_ops USING btree ADD
  -- cross-type comparisons int8 vs int2
  OPERATOR 1 < (int8, int2) ,
  OPERATOR 2 <= (int8, int2) ,
  OPERATOR 3 = (int8, int2) ,
  OPERATOR 4 >= (int8, int2) ,
  OPERATOR 5 > (int8, int2) ,
  FUNCTION 1 btint82cmp(int8, int2) ,

  -- cross-type comparisons int8 vs int4
  OPERATOR 1 < (int8, int4) ,
```

```
OPERATOR 2 <= (int8, int4) ,
OPERATOR 3 = (int8, int4) ,
OPERATOR 4 >= (int8, int4) ,
OPERATOR 5 > (int8, int4) ,
FUNCTION 1 btint84cmp(int8, int4) ,

-- cross-type comparisons int4 vs int2
OPERATOR 1 < (int4, int2) ,
OPERATOR 2 <= (int4, int2) ,
OPERATOR 3 = (int4, int2) ,
OPERATOR 4 >= (int4, int2) ,
OPERATOR 5 > (int4, int2) ,
FUNCTION 1 btint42cmp(int4, int2) ,

-- cross-type comparisons int4 vs int8
OPERATOR 1 < (int4, int8) ,
OPERATOR 2 <= (int4, int8) ,
OPERATOR 3 = (int4, int8) ,
OPERATOR 4 >= (int4, int8) ,
OPERATOR 5 > (int4, int8) ,
FUNCTION 1 btint48cmp(int4, int8) ,

-- cross-type comparisons int2 vs int8
OPERATOR 1 < (int2, int8) ,
OPERATOR 2 <= (int2, int8) ,
OPERATOR 3 = (int2, int8) ,
OPERATOR 4 >= (int2, int8) ,
OPERATOR 5 > (int2, int8) ,
FUNCTION 1 btint28cmp(int2, int8) ,

-- cross-type comparisons int2 vs int4
OPERATOR 1 < (int2, int4) ,
OPERATOR 2 <= (int2, int4) ,
OPERATOR 3 = (int2, int4) ,
OPERATOR 4 >= (int2, int4) ,
OPERATOR 5 > (int2, int4) ,
FUNCTION 1 btint24cmp(int2, int4) ;
```

Заметьте, что в определении семейства «перегружаются» номера стратегий операторов и опорных функций: каждый номер фигурирует в семействе неоднократно. Это допускается, если для каждого экземпляра определённого номера задаются свои типы данных. Экземпляры, у которых оба входных типа совпадают с входным типом класса операторов, являются первичными операторами и опорными функциями для этого класса, и в большинстве случаев они должны объявляться в составе класса операторов, а не быть слабосвязанными членами семейства.

В семействе операторов B-дерева все операторы должны быть совместимыми в контексте сортировки, то есть для всех типов данных, поддерживаемых семейством, должны действовать транзитивные законы: «если A = B и B = C, то A = C», и «если A < B и B < C, то A < C». Более того, неявное или двоичное приведение типов, представленных в семействе операторов, не должно влиять на порядок сортировки. Для каждого оператора в семействе должна существовать опорная функция, принимающая на вход те же два типа, что и оператор. Семейство рекомендуется делать полным, то есть включить в него все операторы для каждого сочетания типов данных. В классы операторов следует включать только однотиповые операторы и опорные функции для определённого типа данных.

Чтобы создать семейство операторов хеширования для нескольких типов данных, необходимо создать совместимые функции поддержки хеша для каждого типа данных, который будет поддерживать семейство. Здесь под совместимостью понимается гарантия получения одного хеш-

кода для любых двух значений, которые операторы сравнения в этом семействе считают равными, даже если они имеют разные типы. Обычно это сложно осуществить, когда типы имеют разное физическое представление, но в некоторых случаях всё же возможно. Более того, преобразование значения одного типа данных, представленного в семействе операторов, к другому типу, также представленному в этом семействе, путём неявного или двоичного сведения не должно менять значение вычисляемого хеша. Заметьте, что единственная опорная функция задаётся для типа данных, а не для оператора равенства. Семейство рекомендуется делать полным, то есть включить в него оператор равенства для всех сочетаний типов данных. В классы операторов следует включать только однотиповый оператор равенства и опорную функция для определённого типа данных.

В индексах GiST, SP-GiST и GIN межтиповые операции явно не выражены. Множество поддерживаемых операторов определяется только теми операциями, которые могут выполнять основные опорные функции заданного класса операторов.

В BRIN требования зависят от инфраструктуры, предоставляющей классы операторов. Для классов операторов, построенных на инфраструктуре `minmax`, требуется то же поведение, что и для семейств операторов B-дерева: все операторы в семействе должны поддерживать совместимый порядок, а приведения не должны влиять на установленный порядок сортировки.

> **Примечание**
>
> До версии 8.3 в PostgreSQL не было понятия семейства операторов, поэтому любые межтиповые операторы, предназначенные для применения с индексом, должны были привязываться непосредственно к классу оператора индекса. Хотя этот подход по-прежнему работает, он считается устаревшим, потому что он создаёт слишком много зависимостей для индекса, а также потому, что планировщик может выполнять межтиповые сравнения более эффективно, когда для обоих типов данных определены операторы в одном семействе.

35.14.6. Системные зависимости от классов операторов

Postgres Pro использует классы операторов для наделения операторов такими свойствами, которые могут быть полезны не только для индексов. Поэтому классы операторов могут быть полезны, даже если вы не намерены индексировать столбцы со значениями определённого вами типа.

В частности, это касается SQL-конструкций `ORDER BY` и `DISTINCT`, для которых требуется сравнивать и упорядочивать значения. Чтобы эти конструкции работали с определённым пользователем типом данных, Postgres Pro задействует класс операторов B-дерева по умолчанию для этого типа. Член «равно» этого класса определяет, как система будет понимать равенство значений для `GROUP BY` и `DISTINCT`, а порядок сортировки, задаваемый классом операторов, определяет порядок `ORDER BY` по умолчанию.

Сравнение массивов пользовательских типов также зависит от семантики, определённой классом операторов B-дерева по умолчанию.

Если класс операторов B-дерева по умолчанию для типа данных не определён, система будет искать класс операторов хеширования по умолчанию. Но так как этот вид класса поддерживает только равенство, на практике его достаточно только для проверки равенства массивов.

Если для типа не определён класс операторов по умолчанию, попытавшись использовать эти конструкции SQL с данным типом, вы получите ошибку вида «не удалось найти оператор сортировки».

> **Примечание**
>
> До версии PostgreSQL 7.4, в операциях сортировки и группировки неявно использовались операторы с именами `=`, `<` и `>`. С новым подходом, опирающимся на классы операторов

по умолчанию, система не делает никаких предположений о поведении операторов по их именам.

Также важно отметить, что оператор, указанный в семействе операторов хеширования, является кандидатом для применения при слиянии и агрегации по хешу, а также при связанной оптимизации. Семейство операторов хеширования играет в данном случае определяющую роль, так как именно в нём задаётся функция хеширования.

35.14.7. Операторы упорядочивания

Некоторые методы доступа индексов (в настоящее время только GiST) поддерживают концепцию *операторов упорядочивания*. Операторы, которые мы обсуждали до этого, были *операторами поиска*. Оператором поиска называется такой оператор, для которого можно выполнить поиск по индексу и найти все строки, удовлетворяющие условию `WHERE` *`индексированный_столбец оператор константа`*. Заметьте, что при этом ничего не говорится о порядке, в котором будут возвращены подходящие строки. Оператор упорядочивания, напротив, не ограничивает набор возвращаемых строк, но определяет их порядок. С таким оператором, просканировав индекс, можно получить строки в порядке, заданным указанием `ORDER BY` *`индексированный_столбец оператор константа`*. Такое определение объясняется тем, что оно поддерживает поиск ближайшего соседа, если этот оператор вычисляет расстояние. Например, запрос

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

находит десять ближайших к заданной точке мест. Индекс GiST по столбцу location может сделать это эффективно, так как `<->` — это оператор упорядочивания.

Тогда как операторы поиска должны возвращать логические результаты, операторы упорядочивания обычно возвращают другой тип, например, float или numeric для расстояний. Этот тип, как правило, отличается от типа индексируемых данных. Чтобы избежать жёстко запрограммированных предположений о поведении различных типов данных, при объявлении оператора упорядочивания должно указываться семейство операторов B-дерева, определяющее порядок сортировки результирующего типа данных. Как было отмечено в предыдущем разделе, семейства операторов B-дерева определяют понятие упорядочивания для Postgres Pro, так что такое объявление оказывается естественным. Так как оператор `<->` для точек возвращает `float8`, его можно включить в команду создания класса операторов так:

```
OPERATOR 15    <-> (point, point) FOR ORDER BY float_ops
```

где `float_ops` — встроенное семейство операторов, включающее операции с `float8`. Это объявление означает, что индекс может возвращать строки в порядке увеличения значений оператора `<->`.

35.14.8. Особенности классов операторов

Есть ещё две особенности классов операторов, которые мы до этого не обсуждали, в первую очередь потому, что они не востребованы для наиболее часто применяемых методов индексов.

Обычно объявление оператора в качестве члена класса операторов (или семейства) означает, что метод индекса может получить точно набор строк, который удовлетворяет условию `WHERE` с этим оператором. Например, запрос:

```
SELECT * FROM table WHERE integer_column < 4;
```

может быть удовлетворён в точности индексом-B-деревом по целочисленному столбцу. Но бывают случаи, когда индекс полезен как приблизительный указатель на соответствующие строки. Например, если индекс GiST хранит только прямоугольники, описанные вокруг геометрических объектов, он не может точно удовлетворить условие `WHERE`, которое проверяет пересечение не прямоугольных объектов, а например, многоугольников. Однако этот индекс можно применить, чтобы найти объекты, для которых описанные вокруг прямоугольники

пересекаются с прямоугольником, описанным вокруг целевого объекта, а затем провести точную проверку пересечения только для найденных по индексу объектов. Если это имеет место, такой индекс называется «неточным» для оператора. Для реализации поиска по неточному индексу метод индекса возвращает флаг *recheck* (перепроверить), когда строка может действительно удовлетворять, а может не удовлетворять условию запроса. Затем исполнитель запроса перепроверяет полученную строку по исходному условию запроса и определяет, должна ли она выдаваться как действительно соответствующая ему. Этот подход работает, если индекс гарантированно выдаёт все требуемые строки плюс, возможно, дополнительные строки, которые можно исключить, вызвав первоначальный оператор. Методы индексов, поддерживающие неточный поиск (в настоящее время, GiST, SP-GiST и GIN), позволяют устанавливать флаг recheck опорным функциям отдельных классов операторов, так что по сути это особенность класса операторов.

Вернёмся к ситуации, когда мы храним в индексе только прямоугольник, описанный вокруг сложного объекта, такого как многоугольник. В этом случае нет большого смысла хранить в элементе индекса весь многоугольник — мы можем с тем же успехом хранить более простой объект типа `box`. Это отклонение выражается указанием `STORAGE` в команде `CREATE OPERATOR CLASS`, которое записывается примерно так:

```
CREATE OPERATOR CLASS polygon_ops
    DEFAULT FOR TYPE polygon USING gist AS
        ...
        STORAGE box;
```

В настоящее время только методы индексов GiST, GIN и BRIN позволяют задать в `STORAGE` тип, отличный от типа данных столбца. В GiST преобразованием данных, связанным с использованием `STORAGE`, должны заниматься опорные процедуры `compress` и `decompress`. В GIN тип `STORAGE` определяет тип значений «ключа», который обычно отличается от типа индексируемого столбца — например, в классе операторов для столбцов с целочисленным массивом ключами могут быть просто целые числа. За извлечение ключей из индексированных значений в GIN отвечают опорные функции `extractValue` и `extractQuery`. BRIN похож на GIN: в нём тип `STORAGE` определяет тип хранимых обобщённых значений, а опорные процедуры классов операторов отвечают за правильное прочтение этих значений.

35.15. Упаковывание связанных объектов в расширение

Полезное расширение Postgres Pro обычно включает несколько объектов SQL; например, с появлением нового типа данных могут потребоваться новые функции, новые операторы и новые классы операторов. Все эти объекты удобно собрать в один пакет, с тем чтобы упростить управление базой данных. В Postgres Pro такие пакеты называются *расширениями*. Чтобы определить расширение, вам понадобится как минимум *файл скрипта* с командами SQL, создающими объекты расширения, и *управляющий файл*, в котором определяются несколько базовых свойств самого расширения. Если расширение написано на C, в него обычно также включается файл разделяемой библиотеки, содержащий скомпилированный код. Обеспечив наличие этих файлов, загрузить их в базу данных можно простой командой CREATE EXTENSION.

Основное преимущество расширений по сравнению с обычным SQL-скриптом, загружающим множество «разрозненных» объектов в базу данных, состоит в том, что Postgres Pro будет понимать, что объекты расширения связаны вместе. Вы можете удалить все объекты одной командой DROP EXTENSION (разрабатывать отдельный скрипт «uninstall» не требуется). Ещё полезнее то, что утилита pg_dump знает, что не нужно выгружать отдельные объекты, составляющие расширение — вместо этого она просто включит в архивный файл команду `CREATE EXTENSION`. Это кардинально упрощает миграцию на новую версию расширения, которая может содержать новые или другие объекты по сравнению с предыдущей версией. Заметьте, однако, что при загрузке такого архива в базу данных обязательно наличие скрипта, управляющего файла и других файлов расширения.

Postgres Pro не позволит удалить отдельный объект, содержащийся в расширении, кроме как при удалении всего расширения. Также имейте в виду, что вы можете изменить определение

объекта, относящегося к расширению (например, командой `CREATE OR REPLACE FUNCTION` для функции), но изменённое определение не будет выгружено утилитой pg_dump. Такие изменения обычно разумны, только если они параллельно отражаются в файле скрипта расширения. (Для таблиц, содержащих конфигурационные данные, предусмотрены специальные средства; см. Подраздел 35.15.3.) В производственной среде обычно лучше создавать скрипт обновления расширения, который будет изменять относящиеся к расширению объекты.

Скрипт расширения может устанавливать права доступа для объектов, являющихся частью расширения, выполняя команды `GRANT` и `REVOKE`. Окончательный набор прав для каждого объекта (если они заданы) будет сохранён в системном каталоге `pg_init_privs`. При использовании pg_dump в выгружаемый скрипт будет выведена команда `CREATE EXTENSION` с последующими операторами `GRANT` и `REVOKE`, которые установят права, имевшие место в момент выгрузки.

Postgres Pro в настоящее время не поддерживает скрипты расширений, выполняющие операторы `CREATE POLICY` или `SECURITY LABEL`. Ожидается, что такие команды будут выполняться после того, как расширение будет создано. Выгружая данные, pg_dump будет также включать в вывод все политики RLS и метки безопасности.

Механизм расширений также предоставляет средства для поддержки дополнительных скриптов, призванных изменять определение объектов SQL, содержащихся в расширении. Например, если версия расширения 1.1, по сравнению с версией 1.0, добавляет одну функцию и изменяет тело другой функции, автор расширения может предоставить *скрипт обновления*, который произведёт именно эти два изменения. Затем, воспользовавшись командой `ALTER EXTENSION UPDATE`, можно будет применить эти изменения и отследить, какая версия расширения фактически установлена в заданной базе данных.

Типы SQL-объектов, которые могут быть членами расширения, перечислены в описании ALTER EXTENSION. Не могут быть его членами, в частности, объекты уровня кластера, такие как базы данных, роли и табличные пространства, так как расширение существует только в рамках одной базы данных. (Скрипту расширения не запрещается создавать такие объекты, но если он сделает это, они не будут считаться частью расширения.) Также заметьте, что несмотря на то, что таблица может быть членом расширения, её подчинённые объекты, такие как индексы, непосредственными членами расширения считаться не будут. Ещё один важный момент — схемы могут принадлежать расширениям, но не наоборот; поэтому расширение имеет неполное имя и не существует «внутри» какой-либо схемы. Однако объекты-члены расширения, будут относиться к схемам, если это уместно для их типов. Сами расширения могут иметь, а могут и не иметь основания владеть схемами, к которым относятся объекты-члены расширения.

Если скрипт расширения создаёт какие-либо временные объекты (например, временные таблицы), эти объекты будут считаться членами расширения до конца текущего сеанса, но удалятся автоматически в конце сеанса, как и должны временные объекты. Это является исключением из правила, запрещающего удаление объектов-членов расширения без удаления всего расширения.

35.15.1. Файлы расширений

Команда CREATE EXTENSION задействует управляющий файл расширения, который должен называться по имени расширения, с суффиксом `.control`, и должен быть помещён в каталог сервера `SHAREDIR/extension`. Должен быть также ещё минимум один SQL-скрипт, с именем, соответствующим шаблону *`расширение--версия`*`.sql` (например, `foo--1.0.sql` для версии `1.0` расширения `foo`). По умолчанию скрипт(ы) также помещается в каталог `SHAREDIR/extension`; но в управляющем файле можно задать и другой каталог.

Формат управляющего файла расширения не отличается от формата `postgresql.conf`, а именно представляет собой список присваиваний *`имя_параметра = значение`*, по одному в строке. В нём также допускаются пустые строки и комментарии, начинающиеся с `#`. Все значения, отличные от единственного слова или числа, в нём должны заключаться в кавычки.

В управляющем файле могут устанавливаться следующие параметры:

`directory` (`string`)

Каталог, содержащий SQL-скрипт(ы) расширения. Если только не задан абсолютный путь, это имя рассматривается относительно каталога сервера `SHAREDIR`. По умолчанию подразумевается указание `directory = 'extension'`.

`default_version` (`string`)

Версия расширения по умолчанию (та, которая будет установлена, если в `CREATE EXTENSION` не будет указана никакая версия). Хотя этот параметр можно опустить, это приведёт к ошибке в `CREATE EXTENSION` без явного указания `VERSION`, что вряд ли будет желаемым поведением.

`comment` (`string`)

Комментарий (произвольная строка) к расширению. Комментарий применяется при изначальном создании расширения, но не при обновлениях расширения (так как при этом мог бы заменяться комментарий, заданный пользователем). Комментарий расширения также можно задать посредством команды COMMENT в файле скрипта.

`encoding` (`string`)

Кодировка символов, используемая в файлах скриптов. Её следует указать, если эти файлы содержат символы не из набора ASCII. По умолчанию предполагается, что эти файлы содержат текст в кодировке базы данных.

`module_pathname` (`string`)

Значение этого параметра будет подставляться вместо каждого вхождения `MODULE_PATHNAME` в скриптах. Если этот параметр не задан, подстановка не производится. Обычно для этого параметра устанавливается значение `$libdir/`*`имя_разделяемой_библиотеки`*, а затем в командах `CREATE FUNCTION` для функций на языке C указывается `MODULE_PATHNAME`, чтобы в скриптах не приходилось жёстко задавать имя разделяемой библиотеки.

`requires` (`string`)

Список имён расширений, от которых зависит данное, например, `requires = 'foo, bar'`. Эти расширения должны быть уже установлены, прежде чем можно будет установить данное.

`superuser` (`boolean`)

Если этот параметр имеет значение `true` (по умолчанию), только суперпользователи смогут создать это расширение или обновить его до новой версии. Если он равен `false`, для этого будет достаточно прав, необходимых для выполнения команд в установочном скрипте или скрипте обновления.

`relocatable` (`boolean`)

Расширение является *перемещаемым*, если относящиеся к нему объекты после создания расширения можно переместить в другую схему. По умолчанию подразумевается `false`, то есть расширение не считается перемещаемым. Подробнее об этом рассказывается в Подразделе 35.15.2.

`schema` (`string`)

Этот параметр может задаваться только для неперемещаемых расширений. Если он задан, расширение можно будет загрузить только в указанную схему и не в какую другую. Подробнее об этом рассказывается ниже. Параметр `schema` учитывается только при изначальном создании расширения, но не при его обновлении. Подробнее об этом рассказывается в Подразделе 35.15.2.

Помимо главного управляющего файла *`расширение`*`.control`, расширение может включать дополнительные управляющие файлы с именами вида *`расширение`*`--`*`версия`*`.control`. Если они присутствуют, они должны находиться в том же каталоге, что и основной скрипт. Дополнительные

управляющие файлы имеют тот же формат, что и основной. Любые параметры, заданные в дополнительном управляющем файле, переопределяют параметры основного файла, когда выполняется установка этой версии расширения или обновление до неё. Однако параметры `directory` и `default_version` в дополнительных управляющих файлах задать нельзя.

SQL-скрипты расширений могут содержать любые команды SQL, за исключением команд управления транзакциями (`BEGIN`, `COMMIT` и т. д.) и команд, которые не могут выполняться внутри блока транзакции (например, `VACUUM`). Это объясняется тем, что эти скрипты неявно выполняются в блоке транзакции.

SQL-скрипты расширений также могут содержать строки, начинающиеся с `\echo`, и они будут игнорироваться (восприниматься как комментарии) механизмом расширений. Это часто используется для вывода ошибки в случае, если этот скрипт выполняется в psql, а не загружается командой `CREATE EXTENSION` (см. пример скрипта в Подразделе 35.15.7). Если такое выполнение не предотвратить, пользователи могут случайно загрузить содержимое расширения как «разрозненные» объекты, а не как собственно расширение, и получить состояние, которое довольно сложно исправить.

Тогда как файлы скриптов могут содержать любые символы, допустимые в указанной кодировке, управляющие файлы могут содержать только ASCII-символы, так как указать кодировку этих файлов в Postgres Pro нет никакой возможности. На практике это представляет проблему, только если вы хотите использовать символы не из набора ASCII в комментарии расширения. В таких случаях рекомендуется не использовать параметр `comment` в управляющем файле, а вместо этого задать комментарий командой `COMMENT ON EXTENSION` в файле скрипта.

35.15.2. Перемещаемость расширений

У пользователей часто возникает желание загрузить объекты, содержащиеся в расширении, в схему, отличную от той, что выбрал автор расширения. Насколько это поддерживает расширение, описывается одним из трёх уровней:

- Полностью перемещаемое расширение может быть перемещено в другую схему в любое время, даже после того, как оно загружено в базу данных. Это осуществляется командой `ALTER EXTENSION SET SCHEMA`, которая автоматически переименовывает все объекты-члены расширения, перенося их в новую схему. Обычно это возможно, только если в расширении нет никаких внутренних предположений о том, в какой схеме находятся все его объекты. Кроме того, все объекты расширения должны находиться в одной исходной схеме (за исключением объектов, не принадлежащих схемам, как например, процедурные языки). Чтобы пометить расширение как полностью перемещаемое, установите `relocatable = true` в его управляющем файле.
- Расширение может быть перемещаемым в момент установки, но не после. Обычно это имеет место, когда скрипту расширения необходимо явно ссылаться на целевую схему, например, устанавливая свойства `search_path` для функций SQL. Для такого расширения нужно задать `relocatable = false` в его управляющем файле и обращаться к целевой схеме в скрипте по псевдоимени `@extschema@`. Все вхождения этого псевдоимени будут заменены именем выбранной целевой схемы перед выполнением скрипта. Пользователь может выбрать целевую схему в указании `SCHEMA` команды `CREATE EXTENSION`.
- Если расширение вовсе не поддерживает перемещение, установите в его управляющем файле `relocatable = false`, и также задайте в параметре `schema` имя предполагаемой целевой схемы. Это предотвратит использование указания `SCHEMA` команды `CREATE EXTENSION`, если только оно задаёт не то же имя, что определёно в управляющем файле. Этот выбор обычно необходим, если в расширении делаются внутренние предположения об именах схемы, которые нельзя свести к использованию псевдоимени `@extschema@`. Механизм подстановки `@extschema@` будет работать и в этом случае, хотя польза от него будет ограниченной, так как имя схемы определяется управляющим файлом.

В любом случае при выполнении файла скрипта параметр search_path изначально будет указывать на целевую схему; то есть, `CREATE EXTENSION` делает то же, что и:

```
SET LOCAL search_path TO @extschema@, pg_temp;
```

Это позволяет направить объекты, создаваемые скриптом, в целевую схему. Скрипт может изменить `search_path`, если пожелает, но обычно это нежелательно. Параметр `search_path` восстанавливает предыдущее значение по завершении `CREATE EXTENSION`.

Целевая схема определяется параметром `schema` (если он задан) в управляющем файле, либо указанием `SCHEMA` команды `CREATE EXTENSION` (если оно присутствует), а в противном случае выбирается текущая схема для создания объектов по умолчанию (первая указанная в параметре `search_path` вызывающего). Когда используется параметр управляющего файла `schema`, целевая схема будет создана, если она ещё не существует, но в двух других случаях она должна уже существовать.

Если в параметре `requires` в управляющем файле расширения указаны какие-либо расширения, необходимые для данного, их целевые схемы добавляются к начальному значению `search_path` после целевой схемы нового расширения. Благодаря этому их объекты видны для скрипта нового расширения.

В целях безопасности схема `pg_temp` всегда автоматически добавляется в конец `search_path`.

Хотя неперемещаемое расширение может содержать объекты, распределяемые по нескольким схемам, обычно желательно поместить все объекты, предназначенные для внешнего использования, в одну схему, назначенную целевой схемой расширения. Такой порядок будет хорошо согласовываться со значением `search_path` по умолчанию в процессе создания зависимых расширений.

35.15.3. Конфигурационные таблицы расширений

Некоторые расширения включают конфигурационные таблицы, содержащие данные, которые могут быть добавлены или изменены пользователем после установки расширения. Обычно, если таблица является частью расширения, ни определение таблицы, ни её содержимое не будет выгружаться утилитой pg_dump. Но это поведение нежелательно для конфигурационных таблиц — изменения, внесённые в них пользователем, должны выгружаться; в противном случае расширение будет вести себя по-другому, когда будет загружено вновь.

Чтобы решить эту проблему, скрипт расширения может пометить созданную им таблицу или последовательность как конфигурационное отношение, в результате чего pg_dump включит в выгружаемые данные содержимое (но не определение) этой таблицы или последовательности. Для этого нужно вызвать функцию `pg_extension_config_dump(regclass, text)` после создания таблицы или последовательности, например так:

```
CREATE TABLE my_config (key text, value text);
CREATE SEQUENCE my_config_seq;

SELECT pg_catalog.pg_extension_config_dump('my_config', '');
SELECT pg_catalog.pg_extension_config_dump('my_config_seq', '');
```

Так можно пометить любое число таблиц или последовательностей, в том числе последовательности, связанные со столбцами `serial` или `bigserial`.

Когда второй аргумент `pg_extension_config_dump` — пустая строка, pg_dump выгружает всё содержимое таблицы. Обычно это правильно, только если после создания скриптом расширения эта таблица изначально пуста. Если же в таблице оказывается смесь начальных данных и данных, добавленных пользователем, во втором аргументе `pg_extension_config_dump` передаётся условие `WHERE`, которое отфильтровывает данные, подлежащие выгрузке. Например, имея таблицу, созданную таким образом:

```
CREATE TABLE my_config (key text, value text, standard_entry boolean);

SELECT pg_catalog.pg_extension_config_dump('my_config', 'WHERE NOT standard_entry');
```

можно сделать так, чтобы поле `standard_entry` содержало true только для строк, создаваемых скриптом расширения.

Для последовательностей второй аргумент функции `pg_extension_config_dump` не имеет значения.

В более сложных ситуациях, когда пользователи могут модифицировать и изначально существовавшие строки, можно создать триггеры для конфигурационной таблицы, которые корректно пометят изменённые строки.

Условие фильтра, связанное с конфигурационной таблицей, можно изменить, повторно вызвав `pg_extension_config_dump`. (Обычно это находит применение в скрипте обновления расширения.) Единственный способ обозначить, что некоторая таблица более не является конфигурационной — разорвать её связь с расширением командой `ALTER EXTENSION ... DROP TABLE`.

Заметьте, что отношения внешних ключей между таблицами определяют порядок, в котором эти таблицы будет выгружать pg_dump. В частности, pg_dump попытается выгрузить сначала основную таблицу, а затем подчинённую. Так как отношения внешних ключей устанавливаются во время выполнения CREATE EXTENSION (до загрузки данных в таблицы), циклические зависимости не поддерживаются. Когда образуются циклические зависимости, данные тем не менее будут выгружены, но полученный архив нельзя будет восстановить обычным образом, потребуется вмешательство пользователя.

Последовательности, связанные со столбцами `serial` или `bigserial`, не обязательно помечать непосредственно, чтобы их состояние было сохранено. Для этой цели достаточно пометить только их родительское отношение.

35.15.4. Обновление расширений

Один из плюсов механизма расширений заключается в том, что он предоставляет удобные способы управления обновлениями SQL-команд, определяющих объекты расширения. В частности, каждой выпускаемой версии установочного скрипта расширения назначается имя или номер версии. Кроме того, если вы хотите, чтобы пользователи могли динамически обновлять одну версию расширения до другой, вы должны предоставить *скрипты обновления*, которые внесут необходимые изменения для перехода от старой версии к новой. Скриптам обновлений назначаются имена, соответствующие шаблону *`расширение--старая_версия--новая_версия`*`.sql` (например, `foo--1.0--1.1.sql` будет содержать команды, меняющие версию `1.0` расширения `foo` на версию `1.1`).

С условием, что имеется подходящий скрипт расширения, команда `ALTER EXTENSION UPDATE` обновит установленное расширение до указанной новой версии. Скрипт обновления запускается в том же окружении, которое организует команда `CREATE EXTENSION` для установочных скриптов: в частности, `search_path` устанавливается таким же образом, а любые новые объекты, созданные скриптом, автоматически добавляются в расширение. И если скрипт решит удалить объекты-члены расширения, они будут автоматически исключены из его состава.

Если у расширения есть дополнительные управляющие файлы, для скрипта обновления применяются те параметры, которые связаны с целевой (новой) версией скрипта.

Механизм обновления может использоваться для решения важной особой задачи: преобразование «разрозненной» коллекции объектов в расширение. До того как механизм расширений появился в PostgreSQL (в версии 9.1), многие писали модули разрешений, которые просто создавали множество неупакованных объектов. Но если у нас уже есть база данных с такими объектами, как их можно преобразовать в правильно упакованное расширение? Один из вариантов — удалить их и затем выполнить простую команду `CREATE EXTENSION`, но это нежелательно, если у объектов есть зависимости (например, если в таблицах есть столбцы типа данных, созданного расширением). Чтобы исправить эту ситуацию, можно создать пустое расширение, затем с помощью команды `ALTER EXTENSION ADD` добавить в него каждый существующий объект, и наконец, создать все новые объекты, которые есть в текущей версии расширения, но отсутствуют в неупакованном варианте. `CREATE EXTENSION` поддерживает этот сценарий предложением `FROM` *`старая_версия`*, с

которым она не будет запускать обычный установочный скрипт для целевой версии, а запустит вместо этого скрипт обновления с именем *расширение--старая_версия--новая_версия*.sql. В качестве имени *старая_версия* автор расширения может выбрать любое фиктивное имя, но обычно задаётся unpackaged. Если у вас несколько предыдущих версий, которые нужно привести к стилю расширения, выберите разные фиктивные имена версий, чтобы различить их.

ALTER EXTENSION также может выполнять последовательности скриптов обновления для получения запрошенной версии. Например, если имеются только скрипты foo--1.0--1.1.sql и foo--1.1--2.0.sql, ALTER EXTENSION будет применять их по порядку, если при установленной версии 1.0 запрошено обновлении до версии 2.0.

Postgres Pro не делает никаких предположений о свойствах имён версий: например, он не знает, следует ли версия 1.1 за 1.0. Он просто сопоставляет имена имеющихся версий и следует пути, который требует применить как можно меньше скриптов обновлений. (Именем версии на самом деле может быть любая строка, которая не содержит -- и при этом не начинается и не заканчивается символом -.)

Иногда бывают полезны скрипты «понижения версии», например, foo--1.1--1.0.sql, которые позволяют откатить изменения, связанные с версией 1.1. Если вы применяете их, учтите, что есть вероятность неожиданного выполнения такого скрипта, если он окажется в кратчайшем пути. Рискованная ситуация возникает при наличии скрипта обновления по «короткому пути», который перепрыгивает через несколько версий, и скрипта понижения версии до начальной точки первого скрипта. В результате может получиться так, что понижение версии с последующим обновлением по короткому пути окажется на несколько шагов короче, чем последовательное повышение версии. Если скрипт понижения версии удаляет какие-либо незаменимые объекты, это может привести к нежелательным результатам.

Чтобы убедиться, что при обновлении не будет выбран нежелательный путь, воспользуйтесь этой командой:

```
SELECT * FROM pg_extension_update_paths('имя_расширения');
```

Она показывает каждую пару различных известных имён версий для указанного расширения, вместе с последовательностью обновления, которая будет выбрана для перехода от одной версии к другой, либо NULL, если путь обновления не находится. Путь выводится в текстовом виде с разделителями --. Если вы предпочитаете формат массива, вы можете применить regexp_split_to_array(path,'--').

35.15.5. Установка расширений скриптами обновления

Расширение, существующее некоторое время, вероятно, будет иметь несколько версий, для которых автору надо будет писать скрипты обновления. Например, если вы выпустили расширение foo версий 1.0, 1.1 и 1.2, у вас должны быть скрипты обновления foo--1.0--1.1.sql и foo--1.1--1.2.sql. До Postgres Pro версии 10 необходимо было также создавать файлы скриптов foo--1.1.sql и foo--1.2.sql, которые устанавливают непосредственно новые версии скриптов; в противном случае их можно было установить, только установив 1.0 и произведя обновление. Это было утомительно и неэффективно, но теперь такой необходимости нет, так как команда CREATE EXTENSION может сама построить цепочку обновлений. Например, если имеются только файлы скриптов foo--1.0.sql, foo--1.0--1.1.sql и foo--1.1--1.2.sql, то запрос на установку версии 1.2 удовлетворяется запуском этих трёх скриптов по очереди. Это не будет отличаться от установки версии 1.0 с последующим обновлением до 1.2. (Как и с командой ALTER EXTENSION UPDATE, при наличии нескольких путей выбирается самый короткий.) Организация скриптов расширения по такой схеме может упростить сопровождение небольших обновлений.

Если вы используете дополнительные (ориентированные на версию) управляющие файлы для расширения, поддерживаемого по такой схеме, имейте в виду, что управляющий файл нужен для каждой версии, даже если для неё нет отдельного скрипта установки, так как этот файл будет определять, как произвести неявное обновление до этой версии. Например, если в файле foo--1.0.control задаётся requires = 'bar', а в других управляющих файлах foo — нет, зависимость расширения от bar будет удалена при обновлении с версии 1.0 до другой.

35.15.6. Замечания о безопасности расширений

Широко распространяемые расширения не должны строить никаких предположений относительно базы данных, в которой они находятся. Таким образом, функции, предоставляемые расширениями, следует писать в безопасном стиле, так, чтобы их нельзя было скомпрометировать в атаках с использованием пути поиска.

Расширение, у которого свойство `superuser` имеет значение true, должно быть также защищено от угроз безопасности, связанных с действиями, которые выполняются при установке и обновлении расширения. Для злонамеренного пользователя не составит большого труда создать объект типа троянского коня, который впоследствии скомпрометирует выполнение неаккуратно написанного скрипта расширения и позволит этому пользователю стать суперпользователем.

Советы по безопасному написанию функций представлены ниже в Подразделе 35.15.6.1, а советы по написанию установочных скриптов — в Подразделе 35.15.6.2.

35.15.6.1. Замечания о безопасности функций в расширениях

Функции, реализованные в расширениях на языках SQL и PL*, подвержены атакам с использованием пути поиска во время выполнения, так как синтаксический разбор этих функций имеет место, когда они выполняются, а не когда создаются.

На странице `CREATE FUNCTION` даётся полезный совет по безопасному написанию функций с характеристикой `SECURITY DEFINER`. Эти приёмы рекомендуется применять и для функций, предоставляемых расширениями, так как подобная функция может вызываться пользователем с расширенными правами.

Если вы не можете оставить в `search_path` только безопасные схемы, считайте, что каждое заданное без схемы имя может быть разрешено в объект, созданный злонамеренным пользователем. Избегайте конструкций, явно зависящих от `search_path`; например, `IN` и `CASE` *`выражение`* `WHEN` всегда выбирают оператор по пути поиска. Вместо них следует использовать конструкции `OPERATOR(`*`схема`*`.=) ANY` и `CASE WHEN` *`выражение`*.

Расширения общего назначения не должны рассчитывать на то, что они устанавливаются в безопасную схему, что означает, что даже ссылаться на собственные объекты с указанием схемы в них небезопасно. Например, если в расширении определена функция `myschema.myfunc(bigint)`, её вызов в виде `myschema.myfunc(42)` можно перехватить, создав специальную функцию `myschema.myfunc(integer)`. Позаботьтесь о том, чтобы типы параметров функций и операторов в точности соответствовали объявленным типам их аргументов, и используйте явные приведения, где это необходимо.

35.15.6.2. Замечания о безопасности скриптов расширений

Скрипт установки или обновления расширения следует защищать от атак, осуществляемых во время выполнения с использованием пути поиска. Если имя объекта, который должен использоваться в скрипте по замыслу автора, может быть разрешено в какой-либо другой объект, компрометация расширения произойдёт сразу либо позже, когда объект расширения будет использоваться.

Команды DDL, например `CREATE FUNCTION` и `CREATE OPERATOR CLASS`, в целом безопасны, но будьте бдительны в отношении команд, в которых фигурируют произвольные запросы и выражения. Например, требуют проверки команды `CREATE VIEW`, а также выражения `DEFAULT` в `CREATE FUNCTION`.

Иногда в скрипте расширения возникает потребность выполнить произвольный SQL, например, чтобы внести в каталог изменения, невозможные через DDL. В этом случае обязательно выполняйте такие команды с безопасным `search_path`; *не* доверяйте пути, установленному при выполнении `CREATE/ALTER EXTENSION`. Для этого рекомендуется временно сменить `search_path` на `'pg_catalog, pg_temp'` и добавить явные указания схемы, в которую устанавливается расширение, везде, где это требуется. (Этот приём также может быть полезен при создании представлений.)

Ссылки на другие расширения крайне сложно полностью обезопасить, отчасти из-за отсутствия понимания, в какой схеме находится другое расширения. Риски уменьшаются, если оба расширения устанавливаются в одну схему, так как в этом случае зловредный объект не может оказаться перед объектами в схеме целевого расширения при используемом во время установки `search_path`. Однако в настоящее время нет механизма, который бы это требовал.

35.15.7. Пример расширения

Здесь представлен полный пример расширения, в котором средствами исключительно SQL реализуется составной тип с двумя элементами, который может сохранить в своих слотах значения любого типа, названные «k» и «v». Для хранения все значения переводятся в текстовый формат (если они имеют другой формат).

Файл скрипта `pair--1.0.sql` выглядит так:

```
-- complain if script is sourced in psql, rather than via CREATE EXTENSION
\echo Use "CREATE EXTENSION pair" to load this file. \quit

CREATE TYPE pair AS ( k text, v text );

CREATE FUNCTION pair(text, text)
RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1, $2)::@extschema@.pair;';

CREATE OPERATOR ~> (LEFTARG = text, RIGHTARG = text, PROCEDURE = pair);

-- "SET search_path" is easy to get right, but qualified names perform better.
CREATE FUNCTION lower(pair)
RETURNS pair LANGUAGE SQL
AS 'SELECT ROW(lower($1.k), lower($1.v))::@extschema@.pair;'
SET search_path = pg_temp;

CREATE FUNCTION pair_concat(pair, pair)
RETURNS pair LANGUAGE SQL
AS 'SELECT ROW($1.k OPERATOR(pg_catalog.||) $2.k,
               $1.v OPERATOR(pg_catalog.||) $2.v)::@extschema@.pair;';
```

Управляющий файл `pair.control` выглядит так:

```
# расширение pair
comment = 'Тип данных для пары ключ/значение'
default_version = '1.0'
# расширение не может быть перемещаемым, так как использует @extschema@
relocatable = false
```

Хотя вам вряд ли понадобится сборочный файл, только для того, чтобы установить эти два файла в нужный каталог, вы можете использовать `Makefile` следующего содержания:

```
EXTENSION = pair
DATA = pair--1.0.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

Этот Makefile опирается на инфраструктуру PGXS, которая описывается в Разделе 35.16. С ним команда `make install` установит управляющий файл и скрипт в правильный каталог, который определит pg_config.

Когда эти файлы будут установлены, выполните команду CREATE EXTENSION, чтобы загрузить объекты в определённую базу данных.

35.16. Инфраструктура сборки расширений

Если вы задумываетесь о распространении ваших модулей расширения Postgres Pro, знайте, что организовать для них портируемую систему сборки может быть довольно сложно. Поэтому инсталляция Postgres Pro включает инфраструктуру сборки расширений, названную PGXS, так что несложные модули расширений можно собрать просто в среде установленного сервера. PGXS предназначена в первую очередь для расширений, написанных на C, хотя её можно применять и для расширения на чистом SQL. Заметьте, что PGXS не претендует на роль универсальной инфраструктуры сборки, способной собрать любой программный объект, взаимодействующий с Postgres Pro; она просто автоматизирует общие правила для сборки простых модулей расширения сервера. Для более сложных пакетов вам придётся разработать собственную систему сборки.

Чтобы использовать инфраструктуру PGXS для вашего расширения, вы должны написать простой сборочный файл. В нём вы должны установить нужные переменные и подключить глобальный сборочный файл PGXS. Следующий пример собирает модуль расширения с именем `isbn_issn`, который включает разделяемую библиотеку, написанную на C, управляющий файл расширения, SQL-скрипт и текстовый файл документации:

```
MODULES = isbn_issn
EXTENSION = isbn_issn
DATA = isbn_issn--1.0.sql
DOCS = README.isbn_issn

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

Последние три строки всегда должны быть такими. Выше в файле вы определяете переменные или добавляете собственные правила для make.

Установите одну из этих трёх переменных, чтобы указать, что будет собрано:

`MODULES`

список объектов разделяемых библиотек, которые должны быть собраны из исходных файлов с одной основой (суффиксы библиотек в этом списке не указываются)

`MODULE_big`

разделяемая библиотека, которая должна быть собрана из нескольких исходных файлов (объектные файлы перечисляются в `OBJS`)

`PROGRAM`

исполняемая программа, которая должна быть собрана (объектные файлы перечисляются в `OBJS`)

Также можно установить следующие переменные:

`EXTENSION`

имена расширений(я); для каждого имени вы должны предоставить файл *расширение*`.control`, который будет установлен в *префикс*`/share/extension`

`MODULEDIR`

подкаталог в каталоге *префикс*`/share`, в который должны устанавливаться файлы DATA и DOCS (если не задан, подразумевается `extension`, если установлена переменная `EXTENSION`, или `contrib` в противном случае)

`DATA`

произвольные файлы, которые должны быть установлены в *префикс*`/share/$MODULEDIR`

DATA_built

произвольные файлы, которые должны быть сначала собраны, а затем установлены в *префикс*/share/$MODULEDIR

DATA_TSEARCH

произвольные файлы, которые должны быть установлены в *префикс*/share/tsearch_data

DOCS

произвольные файлы, которые должны быть установлены в *префикс*/doc/$MODULEDIR

SCRIPTS

скрипты (не двоичные файлы), которые должны быть установлены в *префикс*/bin

SCRIPTS_built

скрипты (не двоичные файлы), которые должны быть сначала собраны, а затем установлены в *префикс*/bin

REGRESS

список тестов регрессий (без суффикса), см. ниже

REGRESS_OPTS

дополнительные параметры, передаваемые pg_regress

NO_INSTALLCHECK

не определять цель installcheck; это полезно, если тестам требуется особая конфигурация или pg_regress не используется

EXTRA_CLEAN

дополнительные файлы, которые должны быть удалены при make clean

PG_CPPFLAGS

флаги, добавляемые перед другими в CPPFLAGS

PG_CFLAGS

флаги, добавляемые в CFLAGS

PG_CXXFLAGS

флаги, добавляемые в CXXFLAGS

PG_LDFLAGS

флаги, добавляемые перед другими в LDFLAGS

PG_LIBS

будет добавлено в строку компоновки PROGRAM

SHLIB_LINK

будет добавлено в строку компоновки MODULE_big

PG_CONFIG

путь к программе pg_config в инсталляции Postgres Pro, с которой будет выполняться сборка (обычно указывается просто pg_config, и используется первый экземпляр, найденный по пути в PATH)

Поместите этот сборочный файл под именем `Makefile` в каталог, где находится ваше расширение. После этого выполните `make`, чтобы скомпилировать, а затем `make install`, чтобы установить ваш модуль. По умолчанию расширение компилируется и устанавливается для той инсталляции Postgres Pro, которая соответствует экземпляру `pg_config`, найденному первым при поиске по пути в `PATH`. Чтобы использовать другую инсталляцию, вы можете задать в `PG_CONFIG` путь к её экземпляру `pg_config` либо внутри сборочного файла, либо в командном файле `make`.

Вы также можете запустить `make` в каталоге вне каталога исходного дерева вашего расширения, если хотите отделить каталог сборки. Эта процедура называется сборкой с *VPATH* и выполняется так:

```
mkdir build_dir
cd build_dir
make -f /path/to/extension/source/tree/Makefile
make -f /path/to/extension/source/tree/Makefile install
```

Также вы можете подготовить каталог для сборки с VPATH таким же образом, как это делается в коде ядра сервера. Как один из вариантов, для этого можно воспользоваться скриптом ядра `config/prep_buildtree`. Затем вы сможете выполнить сборку, установив переменную `VPATH` для `make` таким образом:

```
make VPATH=/path/to/extension/source/tree
make VPATH=/path/to/extension/source/tree install
```

Эта процедура поддерживает самые разные расположения каталогов.

Скрипты, перечисленные в переменной `REGRESS`, используются для регрессионного тестирования модуля, и вызвать их можно командой `make installcheck` после `make install`. Для проведения тестирования необходим работающий сервер Postgres Pro. Файлы скриптов, перечисленные в `REGRESS`, должны размещаться в подкаталоге `sql/` каталога расширения. Эти файлы должны иметь расширение `.sql`, но указывать его в списке `REGRESS` в сборочном файле не нужно. Для каждого теста также должен создаваться файл с ожидаемым выводом в подкаталоге `expected/`, с тем же базовым именем и расширением `.out`. Команда `make installcheck` выполнит каждый тест в psql и сравнит полученный вывод с ожидаемым. Все выявленные различия будут записаны в файл `regression.diffs` в формате команды `diff -c`. Заметьте, что при попытке запустить тест без файла ожидаемого вывода этот тест будет отмечен как «проблемный», поэтому убедитесь, что все такие файлы присутствуют.

> **Подсказка**
>
> Проще всего для этого создать пустые файлы ожидаемого вывода, а затем выполнить тест (при этом, конечно, будут выявлены несоответствия). Изучите полученные файлы результатов, сохранённые в каталоге `results/`, и, если они соответствуют вашим ожиданиям от теста, скопируйте их в `expected/`.

Глава 36. Триггеры

В этой главе содержится общая информация о разработке триггерных функций. Триггерные функции могут быть написаны на большинстве доступных процедурных языков, включая PL/pgSQL (Глава 40), PL/Tcl (Глава 41), PL/Perl (Глава 42) и PL/Python (Глава 43). После прочтения этого раздела, следует обратиться к разделу, посвящённому любимому процедурному языку, чтобы узнать специфические для него детали разработки триггеров.

Триггерные функции можно писать и на C, хотя большинство людей находит, что проще использовать один из процедурных языков. В настоящее время невозможно написать триггерную функцию на чистом SQL.

36.1. Обзор механизма работы триггеров

Триггер является указанием, что база данных должна автоматически выполнить заданную функцию, всякий раз когда выполнен определённый тип операции. Триггеры можно использовать с таблицами (секционированными и обычными), с представлениями и с внешними таблицами.

Для обычных и сторонних таблиц можно определять триггеры, которые будут срабатывать до или после любой из команд `INSERT`, `UPDATE` или `DELETE`; либо один раз для каждой модифицируемой строки, либо один раз для оператора SQL. Триггеры на `UPDATE` можно установить так, чтобы они срабатывали, только когда в предложении `SET` оператора `UPDATE` упоминаются определённые столбцы. Также триггеры могут срабатывать для операторов `TRUNCATE`. Если происходит событие триггера, для обработки этого события в установленный момент времени вызывается функция триггера.

Для представлений триггеры могут быть определены для выполнения вместо операций `INSERT`, `UPDATE` и `DELETE`. Такие триггеры `INSTEAD OF` вызываются единожды для каждой строки, которая должна быть изменена в этом представлении. Именно функция триггера отвечает за то, чтобы произвести необходимые изменения в нижележащих базовых таблицах представления и должным образом возвращать изменённые строки, чтобы они появлялись в представлении. Триггеры для представлений тоже могут быть определены так, что они будут выполняться единожды для всего оператора SQL, до или после операций `INSERT`, `UPDATE` или `DELETE`. Однако такие триггеры срабатывают, только если для представления определён триггер `INSTEAD OF`. В противном случае все операторы, обращающиеся к представлению, должны быть переписаны в виде операторов, обращающихся к нижележащим базовым таблицам, и тогда будут срабатывать триггеры, установленные для этих таблиц.

Триггерная функция должна быть создана до триггера. Она должна быть объявлена без аргументов и возвращать тип `trigger`. (Триггерная функция получает данные на вход посредством специально переданной структуры `TriggerData`, а не в форме обычных аргументов.)

После создания триггерной функции создаётся триггер с помощью CREATE TRIGGER. Одна и та же триггерная функция может быть использована для нескольких триггеров.

Postgres Pro предлагает как *построчные*, так и *операторные* триггеры. В случае построчного триггера триггерная функция вызывается один раз для каждой строки, затронутой оператором, запустившим триггер. Операторный же триггер, напротив, вызывается только один раз при выполнении соответствующего оператора, независимо от количества строк, которые он затрагивает. В частности оператор, который не затрагивает никаких строк, всё равно приведёт к срабатыванию операторного триггера. Эти два типа триггеров также называют триггерами *уровня строк* и триггерами *уровня оператора*, соответственно. Триггеры на `TRUNCATE` могут быть определены только на уровне оператора, а не на уровне строк.

Триггеры также классифицируются в соответствии с тем, срабатывают ли они до, после или вместо операции. Они называются триггерами `BEFORE`, `AFTER` и `INSTEAD OF`, соответственно. Триггеры `BEFORE` уровня оператора срабатывают до того, как оператор начинает делать что-либо, тогда как триггеры `AFTER` уровня оператора срабатывают в самом конце работы оператора. Эти типы

триггеров могут быть определены для таблиц, представлений или сторонних таблиц. Триггеры BEFORE уровня строки срабатывают непосредственно перед обработкой конкретной строки, в то время как триггеры AFTER уровня строки срабатывают в конце работы всего оператора (но до любого из триггеров AFTER уровня оператора). Эти типы триггеров могут определяться только для несекционированных таблиц и сторонних таблиц, но не для представлений. Триггеры INSTEAD OF могут определяться только для представлений и только на уровне строк: они срабатывают для каждой строки сразу после того как строка представления идентифицирована как подлежащая обработке.

Оператор, нацеленный на родительскую таблицу в иерархии наследования или секционирования, не вызывает срабатывания триггеров уровня оператора для задействованных дочерних таблиц; срабатывать будут только такие триггеры для родительской таблицы. Однако если для этих дочерних таблиц установлены триггеры уровня строк, они будут срабатывать.

Если запрос INSERT содержит предложение ON CONFLICT DO UPDATE, возможно совместное применение и триггеров уровня строк BEFORE INSERT, и триггеров уровня строк BEFORE UPDATE, которое отразится в окончательном состоянии изменяемой строки, если в запросе задействуются столбцы EXCLUDED. При этом обращение к EXCLUDED не обязательно должно иметь место в обоих наборах триггеров BEFORE на уровне строк. Следует рассмотреть возможность получения неожиданного результата, когда имеются и триггеры BEFORE INSERT, и BEFORE UPDATE на уровне строки, и они вместе модифицируют добавляемую/изменяемую строку (проблемы возможны, даже если изменения более или менее равнозначные, но при этом не идемпотентные). Заметьте, что триггеры UPDATE уровня оператора вызываются при ON CONFLICT DO UPDATE независимо от того, будут ли изменены какие-либо строки в результате UPDATE (и даже в случае, когда альтернативный путь UPDATE вообще не выбирается). При выполнении запроса INSERT с предложением ON CONFLICT DO UPDATE сначала выполняются триггеры BEFORE INSERT, затем триггеры BEFORE UPDATE, потом триггеры AFTER UPDATE и, наконец, AFTER INSERT (речь идёт о триггерах на уровне операторов).

Триггерные функции, вызываемые триггерами операторов, должны всегда возвращать NULL. Триггерные функции, вызываемые триггерами строк, могут вернуть строку таблицы (значение типа HeapTuple). У триггера уровня строки, срабатывающего до операции, есть следующий выбор:

- Можно вернуть NULL, чтобы пропустить операцию для текущей строки. Это указывает исполнителю запросов, что не нужно выполнять операцию со строкой вызвавшей триггер (вставку, изменение или удаление конкретной строки в таблице).
- Возвращаемая строка для триггеров INSERT или UPDATE будет именно той, которая будет вставлена или обновлена в таблице. Это позволяет триггерной функции изменять вставляемую или обновляемую строку.

Если в триггере BEFORE уровня строки не планируется использовать любой из этих вариантов, то нужно аккуратно вернуть в качестве результата ту же строку, которая была передана на вход (то есть строку NEW для триггеров INSERT и UPDATE, или строку OLD для триггеров DELETE).

Триггер уровня строки INSTEAD OF должен вернуть либо NULL, чтобы указать, что он не модифицирует базовые таблицы представления, либо он должен вернуть строку представления, полученную на входе (строку NEW для операций INSERT и UPDATE или строку OLD для операций DELETE). Отличное от NULL возвращаемое значение сигнализирует, что триггер выполнил необходимые изменения данных в представлении. Это приведёт к увеличению счётчика количества строк, затронутых командой. Для операций INSERT и UPDATE (и только для них) триггер может изменить строку NEW перед тем как её вернуть. В результате будут изменены данные, возвращаемые INSERT RETURNING или UPDATE RETURNING, что полезно, когда представление должно возвращать не те данные, что были получены.

Возвращаемое значение игнорируется для триггеров уровня строки, вызываемых после операции, поэтому они могут возвращать NULL.

Если есть несколько триггеров на одно и то же событие для одной и той же таблицы, то они будут вызываться в алфавитном порядке по имени триггера. Для триггеров BEFORE и INSTEAD OF потенциально изменённая строка, возвращаемая одним триггером, становится входящей

строкой для следующего триггера. Если любой из триггеров BEFORE или INSTEAD OF возвращает NULL, операция для этой строки прекращается и последующие триггеры (для этой строки) не срабатывают.

В определении триггера можно указать логическое условие WHEN, которое будет проверяться, чтобы посмотреть, нужно ли запускать триггер. В триггерах уровня строки в условии WHEN можно проверять старые и/или новые значения столбцов строки. (В триггерах уровня оператора также можно использовать условие WHEN, хотя в этом случае это не так полезно.) В триггерах BEFORE условие WHEN вычисляется непосредственно перед тем, как триггерная функция будет выполнена, поэтому использование WHEN существенно не отличается от выполнения той же проверки в самом начале триггерной функции. Однако в триггерах AFTER условие WHEN вычисляется сразу после обновления строки и от этого зависит, будет ли поставлено в очередь событие запуска триггера в конце оператора или нет. Поэтому, когда условие WHEN в триггере AFTER не возвращает истину, не требуется ни постановка события в очередь, ни повторная выборка этой строки в конце оператора. Это может существенно ускорить работу операторов, изменяющих большое количество строк, с триггером, который должен сработать только для нескольких. В триггерах INSTEAD OF не поддерживается использование условий WHEN.

Как правило, триггеры BEFORE уровня строки используются для проверки или модификации данных, которые будут вставлены или изменены. Например, триггер BEFORE можно использовать для вставки текущего времени в столбец timestamp или проверки, что два элемента строки согласованы между собой. Триггеры AFTER уровня строки наиболее разумно использовать для каскадного обновления данных в других таблицах или проверки согласованности сделанных изменений с данными в других таблицах. Причина для такого разделения работы в том, что триггер AFTER видит окончательное значение строки, в то время как для триггера BEFORE это не так, ведь могут быть другие триггеры BEFORE, которые сработают позже. Если нет особых причин для выбора между триггерами BEFORE или AFTER, то триггер BEFORE предпочтительнее, так как не требует сохранения информации об операции до конца работы оператора.

Если триггерная функция выполняет команды SQL, эти команды могут заново запускать триггеры. Это известно как каскадные триггеры. Прямых ограничений на количество каскадных уровней не существует. Вполне возможно, что каскадные вызовы приведут к рекурсивному срабатыванию одного и того же триггера. Например, в триггере INSERT может выполняться команда, которая добавляет строку в эту же таблицу, тем самым опять вызывая триггер на INSERT. Обязанность программиста не допускать бесконечную рекурсию в таких случаях.

При определении триггера можно указывать аргументы. Цель включения аргументов в определение триггера в том, чтобы позволить разным триггерам с аналогичными требованиями вызывать одну и ту же функцию. В качестве примера можно создать обобщенную триггерную функцию, которая принимает два аргумента с именами столбцов и записывает текущего пользователя в первый аргумент и текущий штамп времени во второй. При правильном написании такая триггерная функция будет независима от конкретной таблицы, для которой она будет запускаться. Таким образом, одна и та же функция может использоваться при выполнении INSERT в любую таблицу с соответствующими столбцами, чтобы, например, автоматически отслеживать создание записей в транзакционной таблице. Для триггеров UPDATE аргументы также могут использоваться для отслеживания последних сделанных изменений.

У каждого языка программирования, поддерживающего триггеры, есть свой собственный метод доступа из триггерной функции к входным данным триггера. Входные данные триггера включают в себя тип события (например, INSERT или UPDATE), а также любые аргументы, перечисленные в CREATE TRIGGER. Для триггеров уровня строки входные данные также включают строку NEW для триггеров INSERT и UPDATE и/или строку OLD для триггеров UPDATE и DELETE.

Триггеры уровня оператора по умолчанию не имеют возможностей для проверки отдельных строк, модифицированных оператором. Но триггер AFTER STATEMENT может запросить создание для него *переходных таблиц*, чтобы ему были доступны наборы затрагиваемых операцией строк. Триггерам AFTER ROW также могут предоставляться переходные таблицы, чтобы они могли видеть все изменения в таблице, а не только изменения в отдельных строках, для которых они

срабатывают. Метод обращения к переходным таблицам определяется применяемым языком программирования, но обычно переходные таблицы представляются как временные таблицы только для чтения, к которым в триггерной функции можно обращаться, выполняя SQL-команды.

36.2. Видимость изменений в данных

Если в триггерной функции выполняются SQL-команды и эти команды обращаются к таблице, на которую создан триггер, то необходимо знать правила видимости данных, потому что они определяют, будут ли видеть эти SQL-команды изменения в данных, для которых сработал триггер. Кратко:

- Триггеры уровня оператора следуют простым правилам видимости: никакие из изменений, произведённых оператором, не видны в триггерах `BEFORE`, тогда как в триггерах `AFTER` видны все изменения.
- Изменение данных (вставка, обновление или удаление), заставляющее сработать триггер, *не видно* для команд SQL, выполняемых в триггере `BEFORE` уровня строки, потому что это изменение ещё не произошло.
- Тем не менее команды SQL, выполняемые в триггере `BEFORE` уровня строки, *будут* видеть изменения данных в строках, которые уже были обработаны в этом операторе. Это требует осторожности, так как порядок обработки строк в целом непредсказуемый; команда SQL, обрабатывающая множество строк, может делать это в любом порядке.
- Аналогично, триггер `INSTEAD OF` уровня строки увидит изменения данных, внесённые при предыдущих вызовах триггера `INSTEAD OF` для этой же внешней команды.
- Когда срабатывает триггер `AFTER` уровня строки, все изменения сделанные оператором уже выполнены и видны в вызываемой триггерной функции.

Если триггерная функция написана на одном из стандартных процедурных языков, вышеприведённые утверждения применимы, только если функция объявлена как `VOLATILE`. Функции объявленные как `STABLE` или `IMMUTABLE` в любом случае не будут видеть изменений, сделанных вызывающим оператором.

Дополнительную информацию о правилах видимости данных можно найти в Разделе 44.4. Пример в Разделе 36.4 содержит демонстрацию этих правил.

36.3. Триггерные функции на языке C

В этом разделе описываются низкоуровневые детали интерфейса для триггерных функций. Эта информация необходима только при разработке триггерных функций на языке C. При использовании языка более высокого уровня эти детали обрабатываются не видны. В большинстве случаев стоит рассмотреть возможность использования процедурного языка, прежде чем начать разрабатывать триггеры на C. В документации по каждому процедурному языку объясняется, как создавать триггеры на этом языке.

Триггерные функции должны использовать интерфейс функций «версии 1».

Когда функция вызывается диспетчером триггеров, ей не передаются обычные аргументы, но передаётся указатель «context», ссылающийся на структуру `TriggerData`. Функции на C могут проверить, вызваны ли они диспетчером триггеров или нет, выполнив макрос:

```
CALLED_AS_TRIGGER(fcinfo)
```

который разворачивается в:

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

Если возвращается истина, то `fcinfo->context` можно безопасно привести к типу `TriggerData *` и использовать указатель на структуру `TriggerData`. Функция *не* должна изменять структуру `TriggerData` или любые данные, которые на неё указывают.

`struct TriggerData` определяется в `commands/trigger.h`:

```
typedef struct TriggerData
{
    NodeTag          type;
    TriggerEvent     tg_event;
    Relation         tg_relation;
    HeapTuple        tg_trigtuple;
    HeapTuple        tg_newtuple;
    Trigger         *tg_trigger;
    Buffer           tg_trigtuplebuf;
    Buffer           tg_newtuplebuf;
    Tuplestorestate *tg_oldtable;
    Tuplestorestate *tg_newtable;
} TriggerData;
```

где элементы определяются следующим образом:

`type`

Всегда `T_TriggerData`.

`tg_event`

Описывает событие, для которого вызывается функция. Можно использовать следующие макросы для получения информации о `tg_event`:

`TRIGGER_FIRED_BEFORE(tg_event)`

Возвращает истину, если триггер сработал до операции.

`TRIGGER_FIRED_AFTER(tg_event)`

Возвращает истину, если триггер сработал после операции.

`TRIGGER_FIRED_INSTEAD(tg_event)`

Возвращает истину, если триггер сработал вместо операции.

`TRIGGER_FIRED_FOR_ROW(tg_event)`

Возвращает истину, если триггер сработал на уровне строки.

`TRIGGER_FIRED_FOR_STATEMENT(tg_event)`

Возвращает истину, если триггер сработал на уровне оператора.

`TRIGGER_FIRED_BY_INSERT(tg_event)`

Возвращает истину, если триггер сработал для операции `INSERT`.

`TRIGGER_FIRED_BY_UPDATE(tg_event)`

Возвращает истину, если триггер сработал для операции `UPDATE`.

`TRIGGER_FIRED_BY_DELETE(tg_event)`

Возвращает истину, если триггер сработал для операции `DELETE`.

`TRIGGER_FIRED_BY_TRUNCATE(tg_event)`

Возвращает истину, если триггер сработал для операции `TRUNCATE`.

`tg_relation`

Указатель на структуру, описывающую таблицу, для которой сработал триггер. Подробнее об этой структуре в `utils/rel.h`. Самое интересное здесь это `tg_relation->rd_att` (дескриптор записей таблицы) и `tg_relation->rd_rel->relname` (имя таблицы; имеет тип `NameData`, а не `char*`; используйте `SPI_getrelname(tg_relation)`, чтобы получить тип `char*` если потребуется копия имени).

`tg_trigtuple`

Указатель на строку, для которой сработал триггер. Это строка, которая вставляется, обновляется или удаляется. При срабатывании триггера для `INSERT` или `DELETE` это значение нужно вернуть из функции, только если не планируется изменять строку (в случае `INSERT`) или пропускать операцию для этой строки.

`tg_newtuple`

Для триггера на `UPDATE` это указатель на новую версию строки либо `NULL`, если триггер на `INSERT` или `DELETE`. Это значение нужно вернуть из функции в случае `UPDATE`, если не планируется изменять строку или пропускать операцию для этой строки.

`tg_trigger`

Указатель на структуру с типом `Trigger`, определённую в `utils/reltrigger.h`:

```
typedef struct Trigger
{
    Oid         tgoid;
    char       *tgname;
    Oid         tgfoid;
    int16       tgtype;
    char        tgenabled;
    bool        tgisinternal;
    Oid         tgconstrrelid;
    Oid         tgconstrindid;
    Oid         tgconstraint;
    bool        tgdeferrable;
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgnattr;
    int16      *tgattr;
    char      **tgargs;
    char       *tgqual;
    char       *tgoldtable;
    char       *tgnewtable;
} Trigger;
```

где `tgname` — имя триггера, `tgnargs` — количество аргументов в `tgargs`, и `tgargs` — массив указателей на аргументы, указанные в команде `CREATE TRIGGER`. Остальные члены структуры предназначены для внутреннего использования.

`tg_trigtuplebuf`

Буфер, содержащий `tg_trigtuple`, или содержащий `InvalidBuffer` — если нет такой строки или она не хранится в дисковом буфере.

`tg_newtuplebuf`

Буфер, содержащий `tg_newtuple`, или содержащий `InvalidBuffer` — если нет такой строки или она не хранится в дисковом буфере.

`tg_oldtable`

Указатель на структуру типа `Tuplestorestate`, содержащую ноль или несколько строк в формате, определяемом содержимым `tg_relation`, или указатель `NULL`, если переходное отношение `OLD TABLE` отсутствует.

`tg_newtable`

Указатель на структуру типа `Tuplestorestate`, содержащую ноль или несколько строк в формате, определяемом содержимым `tg_relation`, или указатель `NULL`, если переходное отношение `NEW TABLE` отсутствует.

Чтобы обращаться к переходным таблицам в запросах, выполняемых через SPI, используйте SPI_register_trigger_data.

Триггерная функция должна возвращать указатель HeapTuple или указатель NULL (но *не* SQL значение null, то есть не нужно устанавливать *isNull* в истину). Не забудьте, что если не планируете менять обрабатываемую триггером строку, то нужно вернуть либо tg_trigtuple, либо tg_newtuple.

36.4. Полный пример триггера

Вот очень простой пример триггерной функции, написанной на C. (Примеры триггеров для процедурных языков могут быть найдены в документации на процедурные языки.)

Функция trigf сообщает количество строк в таблице ttest и пропускает операцию для строки при попытке вставить пустое значение в столбец x. (Таким образом, триггер действует как ограничение NOT NULL, но не прерывает транзакцию.)

Вначале определение таблицы:

```
CREATE TABLE ttest (
    x integer
);
```

Теперь исходный код триггерной функции:

```
#include "postgres.h"
#include "fmgr.h"
#include "executor/spi.h"       /* this is what you need to work with SPI */
#include "commands/trigger.h"   /* ... triggers ... */
#include "utils/rel.h"          /* ... and relations */

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc   tupdesc;
    HeapTuple   rettuple;
    char       *when;
    bool        checknull = false;
    bool        isnull;
    int         ret, i;

    /* make sure it's called as a trigger at all */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: not called by trigger manager");

    /* tuple to return to executor */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettuple = trigdata->tg_newtuple;
    else
        rettuple = trigdata->tg_trigtuple;

    /* check for null values */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checknull = true;
```

```
    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        when = "before";
    else
        when = "after ";

    tupdesc = trigdata->tg_relation->rd_att;

    /* connect to SPI manager */
    if ((ret = SPI_connect()) < 0)
        elog(ERROR, "trigf (fired %s): SPI_connect returned %d", when, ret);

    /* get number of rows in table */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);

    if (ret < 0)
        elog(ERROR, "trigf (fired %s): SPI_exec returned %d", when, ret);

    /* count(*) returns int8, so be careful to convert */
    i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                    SPI_tuptable->tupdesc,
                                    1,
                                    &isnull));

    elog (INFO, "trigf (fired %s): there are %d rows in ttest", when, i);

    SPI_finish();

    if (checknull)
    {
        SPI_getbinval(rettuple, tupdesc, 1, &isnull);
        if (isnull)
            rettuple = NULL;
    }

    return PointerGetDatum(rettuple);
}
```

После компиляции исходного кода (см. Подраздел 35.9.5) объявляем функцию и триггеры:

```
CREATE FUNCTION trigf() RETURNS trigger
    AS 'filename'
    LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();
```

Теперь можно проверить работу триггера:

```
=> INSERT INTO ttest VALUES (NULL);

INFO:  trigf (fired before): there are 0 rows in ttest
INSERT 0 0

-- Вставка записи пропущена (значение NULL), поэтому триггер AFTER не сработал
```

```
=> SELECT * FROM ttest;
 x
---
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (fired before): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 1 rows in ttest
                                               ^^^^^^^
                                    вспомните, что говорилось о видимости
INSERT 167793 1
vac=> SELECT * FROM ttest;
 x
---
 1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
                                               ^^^^^^^
                                    вспомните, что говорилось о видимости
INSERT 167794 1
=> SELECT * FROM ttest;
 x
---
 1
 2
(2 rows)

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
UPDATE 1
vac=> SELECT * FROM ttest;
 x
---
 1
 4
(2 rows)

=> DELETE FROM ttest;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
                                               ^^^^^^^
                                    вспомните, что говорилось о видимости
DELETE 2
=> SELECT * FROM ttest;
 x
---
(0 rows)
```

Более сложные примеры можно найти в `src/test/regress/regress.c` и в spi.

Глава 37. Триггеры событий

В дополнение к триггерам, рассмотренным в Главе 36, Postgres Pro также предоставляет триггеры событий. В отличие от обычных триггеров, которые подключаются к конкретной таблице и работают только с командами DML, триггеры событий определяются на уровне базы данных и работают с командами DDL.

Как и обычные триггеры, триггеры событий можно создавать на любом процедурном языке, поддерживающим триггеры событий, а также на C, но не на чистом SQL.

37.1. Обзор механизма работы триггеров событий

Триггер события срабатывает всякий раз, когда в базе данных, в которой он определён, происходит связанное с ним событие. В настоящий момент поддерживаются следующие события: `ddl_command_start`, `ddl_command_end`, `table_rewrite` и `sql_drop`. Поддержка дополнительных событий может быть добавлена в будущих выпусках.

Событие `ddl_command_start` происходит непосредственно перед выполнением команд `CREATE`, `ALTER`, `DROP`, `SECURITY LABEL`, `COMMENT`, `GRANT` и `REVOKE`. Проверка на существование объекта перед срабатыванием триггера не производится. В качестве исключения, однако, это событие не происходит для команд DDL, обращающихся к общим объектам кластера базы данных — базам данных, табличным пространствам, ролям, а также к самим триггерам событий. Событие `ddl_command_start` также происходит непосредственно перед выполнением команды `SELECT INTO`, так как она равнозначна команде `CREATE TABLE AS`.

Событие `ddl_command_end` происходит непосредственно после выполнения команд из того же набора. Чтобы получить дополнительную информацию об операциях DDL, повлёкших произошедшее событие, вызовите функцию `pg_event_trigger_ddl_commands()`, возвращающую множество, из кода обработчика события `ddl_command_end` (см. Раздел 9.28). Заметьте, что этот триггер срабатывает после того, как эти действия имели место (но до фиксации транзакции), так что в системных каталогах можно увидеть уже изменённое состояние.

Событие `sql_drop` происходит непосредственно перед событием `ddl_command_end` для команд, которые удаляют объекты базы данных. Для получения списка удалённых объектов используйте возвращающую набор строк функцию `pg_event_trigger_dropped_objects()` в триггере события `sql_drop` (см. Раздел 9.28). Обратите внимание, что триггер выполняется после удаления объектов из таблиц системного каталога, поэтому их невозможно больше увидеть.

Событие `table_rewrite` происходит только после того, как таблица будет перезаписана в результате определённых действий команд `ALTER TABLE` и `ALTER TYPE`. Хотя перезапись таблицы может быть вызвана и другими управляющими операторами, в частности `CLUSTER` и `VACUUM`, событие `table_rewrite` для них не вызывается.

Триггеры событий (как и прочие функции) не могут выполняться в прерванной транзакции. Поэтому, если команда DDL завершается ошибкой, соответствующие триггеры `ddl_command_end` не сработают. И наоборот, если триггер `ddl_command_end` завершился с ошибкой, последующие триггеры событий не сработают, так же как и сама команда не будет выполняться. Похожим образом, если триггер `ddl_command_end` завершится ошибкой, действие команды DDL будет отменено, как это происходит при возникновении ошибки внутри транзакции.

Полный список команд, которые поддерживаются триггерами событий, можно найти в Разделе 37.2.

Для создания триггера события используется команда CREATE EVENT TRIGGER. Предварительно нужно создать функцию, со специальным возвращаемым типом `event_trigger`. Данная функция не обязана возвращать значение (и может не возвращать). Возвращаемый тип служит лишь указанием на то, что функция будет вызываться из триггера события.

Если есть несколько триггеров на одно и то же событие, то они будут вызываться в алфавитном порядке по имени триггера.

В определении триггера можно использовать условие `WHEN`, чтобы, например, триггер `ddl_command_start` срабатывал только для отдельных команд, которые нужно перехватить. Триггеры событий часто используются для ограничения диапазона DDL-команд, доступных пользователям.

37.2. Матрица срабатывания триггеров событий

В Таблице 37.1 перечислены команды, для которых поддерживаются триггеры событий.

Таблица 37.1. Поддержка триггеров событий командами DDL

Тег команды	`ddl_command_start`	`ddl_command_end`	`sql_drop`	`table_rewrite`	**Замечания**
`ALTER AGGREGATE`	X	X	–	–	
`ALTER COLLATION`	X	X	–	–	
`ALTER CONVERSION`	X	X	–	–	
`ALTER DOMAIN`	X	X	–	–	
`ALTER DEFAULT PRIVILEGES`	X	X	–	–	
`ALTER EXTENSION`	X	X	–	–	
`ALTER FOREIGN DATA WRAPPER`	X	X	–	–	
`ALTER FOREIGN TABLE`	X	X	X	–	
`ALTER FUNCTION`	X	X	–	–	
`ALTER LANGUAGE`	X	X	–	–	
`ALTER LARGE OBJECT`	X	X	–	–	
`ALTER MATERIALIZED VIEW`	X	X	–	–	
`ALTER OPERATOR`	X	X	–	–	
`ALTER OPERATOR CLASS`	X	X	–	–	
`ALTER OPERATOR FAMILY`	X	X	–	–	
`ALTER POLICY`	X	X	–	–	
`ALTER PUBLICATION`	X	X	–	–	
`ALTER SCHEMA`	X	X	–	–	

Тег команды	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	**Замечания**
ALTER SEQUENCE	X	X	–	–	
ALTER SERVER	X	X	–	–	
ALTER STATISTICS	X	X	–	–	
ALTER SUBSCRIPTION	X	X	–	–	
ALTER TABLE	X	X	X	X	
ALTER TEXT SEARCH CONFIGURATION	X	X	–	–	
ALTER TEXT SEARCH DICTIONARY	X	X	–	–	
ALTER TEXT SEARCH PARSER	X	X	–	–	
ALTER TEXT SEARCH TEMPLATE	X	X	–	–	
ALTER TRIGGER	X	X	–	–	
ALTER TYPE	X	X	–	X	
ALTER USER MAPPING	X	X	–	–	
ALTER VIEW	X	X	–	–	
COMMENT	X	X	–	–	Только для локальных объектов
CREATE ACCESS METHOD	X	X	–	–	
CREATE AGGREGATE	X	X	–	–	
CREATE CAST	X	X	–	–	
CREATE COLLATION	X	X	–	–	
CREATE CONVERSION	X	X	–	–	
CREATE DOMAIN	X	X	–	–	
CREATE EXTENSION	X	X	–	–	
CREATE FOREIGN DATA WRAPPER	X	X	–	–	
CREATE FOREIGN TABLE	X	X	–	–	

Тег команды	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	**Замечания**
CREATE FUNCTION	X	X	-	-	
CREATE INDEX	X	X	-	-	
CREATE LANGUAGE	X	X	-	-	
CREATE MATERIALIZED VIEW	X	X	-	-	
CREATE OPERATOR	X	X	-	-	
CREATE OPERATOR CLASS	X	X	-	-	
CREATE OPERATOR FAMILY	X	X	-	-	
CREATE POLICY	X	X	-	-	
CREATE PUBLICATION	X	X	-	-	
CREATE RULE	X	X	-	-	
CREATE SCHEMA	X	X	-	-	
CREATE SEQUENCE	X	X	-	-	
CREATE SERVER	X	X	-	-	
CREATE STATISTICS	X	X	-	-	
CREATE SUBSCRIPTION	X	X	-	-	
CREATE TABLE	X	X	-	-	
CREATE TABLE AS	X	X	-	-	
CREATE TEXT SEARCH CONFIGURATION	X	X	-	-	
CREATE TEXT SEARCH DICTIONARY	X	X	-	-	
CREATE TEXT SEARCH PARSER	X	X	-	-	
CREATE TEXT SEARCH TEMPLATE	X	X	-	-	
CREATE TRIGGER	X	X	-	-	
CREATE TYPE	X	X	-	-	

Тег команды	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	**Замечания**
CREATE USER MAPPING	X	X	-	-	
CREATE VIEW	X	X	-	-	
DROP ACCESS METHOD	X	X	X	-	
DROP AGGREGATE	X	X	X	-	
DROP CAST	X	X	X	-	
DROP COLLATION	X	X	X	-	
DROP CONVERSION	X	X	X	-	
DROP DOMAIN	X	X	X	-	
DROP EXTENSION	X	X	X	-	
DROP FOREIGN DATA WRAPPER	X	X	X	-	
DROP FOREIGN TABLE	X	X	X	-	
DROP FUNCTION	X	X	X	-	
DROP INDEX	X	X	X	-	
DROP LANGUAGE	X	X	X	-	
DROP MATERIALIZED VIEW	X	X	X	-	
DROP OPERATOR	X	X	X	-	
DROP OPERATOR CLASS	X	X	X	-	
DROP OPERATOR FAMILY	X	X	X	-	
DROP OWNED	X	X	X	-	
DROP POLICY	X	X	X	-	
DROP PUBLICATION	X	X	X	-	
DROP RULE	X	X	X	-	
DROP SCHEMA	X	X	X	-	
DROP SEQUENCE	X	X	X	-	
DROP SERVER	X	X	X	-	
DROP STATISTICS	X	X	X	-	
DROP SUBSCRIPTION	X	X	X	-	
DROP TABLE	X	X	X	-	

Тег команды	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	**Замечания**
DROP TEXT SEARCH CONFIGURATION	X	X	X	–	
DROP TEXT SEARCH DICTIONARY	X	X	X	–	
DROP TEXT SEARCH PARSER	X	X	X	–	
DROP TEXT SEARCH TEMPLATE	X	X	X	–	
DROP TRIGGER	X	X	X	–	
DROP TYPE	X	X	X	–	
DROP USER MAPPING	X	X	X	–	
DROP VIEW	X	X	X	–	
GRANT	X	X	–	–	Только для локальных объектов
IMPORT FOREIGN SCHEMA	X	X	–	–	
REFRESH MATERIALIZED VIEW	X	X	–	–	
REVOKE	X	X	–	–	Только для локальных объектов
SECURITY LABEL	X	X	–	–	Только для локальных объектов
SELECT INTO	X	X	–	–	

37.3. Триггерные функции событий на языке C

В этом разделе описываются низкоуровневые детали интерфейса для событийных триггерных функций. Эта информация необходима только при разработке событийных триггерных функций событий на языке C. При использовании языка более высокого уровня эти детали не видны. В большинстве случаев стоит рассмотреть возможность использования процедурного языка, прежде чем начать разрабатывать событийные триггеры на C. В документации по каждому процедурному языку объясняется, как создавать событийные триггеры на этом языке.

Триггерные функции событий должны использовать «version 1» интерфейса диспетчера функций.

Когда функция вызывается диспетчером триггеров событий, ей не передаются обычные аргументы, но передаётся указатель «context», ссылающийся на структуру `EventTriggerData`. Функции на C могут проверить вызваны ли они диспетчером триггеров событий или нет выполнив макрос:

```
CALLED_AS_EVENT_TRIGGER(fcinfo)
```

который разворачивается в:

`EventTriggerData`

Если возвращается истина, то `fcinfo->context` можно безопасно привести к типу `EventTriggerData *` и использовать указатель на структуру `EventTriggerData`. Функция *не* должна изменять структуру `EventTriggerData` или любые данные, которые на неё указывают.

`struct EventTriggerData` определена в `commands/event_trigger.h`:

```
typedef struct EventTriggerData
{
    NodeTag     type;
    const char *event;      /* имя события */
    Node       *parsetree;  /* дерево разбора */
    const char *tag;        /* тег команды */
} EventTriggerData;
```

со следующими членами структуры:

`type`

Всегда `T_EventTriggerData`.

`event`

Описывает событие, для которого вызывается функция. Возможные значения: `"ddl_command_start"`, `"ddl_command_end"`, `"sql_drop"`, `"table_rewrite"`. Суть этих событий описывается в Разделе 37.1.

`parsetree`

Указатель на дерево разбора команды. Детали можно посмотреть в исходном коде Postgres Pro. Структура дерева разбора может быть изменена без предупреждений.

`tag`

Тег команды, для которой сработал триггер события. Например `"CREATE FUNCTION"`.

Функция триггера события должна возвращать указатель `NULL` (но *не* SQL значение null, то есть не нужно устанавливать *`isNull`* в истину).

37.4. Полный пример триггера события

Вот очень простой пример функции для триггера события, написанной на C. (Примеры триггеров для процедурных языков могут быть найдены в документации на процедурные языки.)

Функция `noddl` выдаёт ошибку при каждом вызове. Триггер с этой функцией определяется для события `ddl_command_start`. Это предотвращает работу любых DDL-команд (за исключением тех, о которых говорилось в Разделе 37.1).

Теперь исходный код триггерной функции:

```
#include "postgres.h"
#include "commands/event_trigger.h"


PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(noddl);

Datum
noddl(PG_FUNCTION_ARGS)
{
```

```
    EventTriggerData *trigdata;

    if (!CALLED_AS_EVENT_TRIGGER(fcinfo))  /* internal error */
        elog(ERROR, "not fired by event trigger manager");

    trigdata = (EventTriggerData *) fcinfo->context;

    ereport(ERROR,
        (errcode(ERRCODE_INSUFFICIENT_PRIVILEGE),
                 errmsg("command \"%s\" denied", trigdata->tag)));

    PG_RETURN_NULL();
}
```

После компиляции исходного кода (см. Подраздел 35.9.5) объявляем функцию и триггеры:

```
CREATE FUNCTION noddl() RETURNS event_trigger
    AS 'noddl' LANGUAGE C;

CREATE EVENT TRIGGER noddl ON ddl_command_start
    EXECUTE PROCEDURE noddl();
```

Теперь проверим работу триггера:

```
=# \dy
                      List of event triggers
 Name  |       Event       | Owner | Enabled | Procedure | Tags
-------+-------------------+-------+---------+-----------+------
 noddl | ddl_command_start | dim   | enabled | noddl     |
(1 row)

=# CREATE TABLE foo(id serial);
ERROR:  Команда "CREATE TABLE" отменена
```

В этой ситуации, для запуска DDL-команд, нужно либо удалить триггер события, либо отключить его. Может быть удобным отключить триггер на время выполнения транзакции:

```
BEGIN;
ALTER EVENT TRIGGER noddl DISABLE;
CREATE TABLE foo (id serial);
ALTER EVENT TRIGGER noddl ENABLE;
COMMIT;
```

(Вспомним, что триггеры событий не обрабатывают DDL-команды для самих триггеров событий.)

37.5. Пример событийного триггера, обрабатывающего перезапись таблицы

Благодаря существованию события `table_rewrite`, можно реализовать политику перезаписи таблиц, допускающую перезапись только в определённое время обслуживания.

Следующий пример демонстрирует реализацию такой политики.

```
CREATE OR REPLACE FUNCTION no_rewrite()
 RETURNS event_trigger
 LANGUAGE plpgsql AS
$$
---
--- Реализация локальной политики перезаписи таблиц:
---   перезапись public.foo не допускается,
---   другие таблицы могут перезаписываться между 1 часом ночи и 6 часами утра,
```

```
---   если только их размер не превышает 100 блоков
---
DECLARE
  table_oid oid := pg_event_trigger_table_rewrite_oid();
  current_hour integer := extract('hour' from current_time);
  pages integer;
  max_pages integer := 100;
BEGIN
  IF pg_event_trigger_table_rewrite_oid() = 'public.foo'::regclass
  THEN
        RAISE EXCEPTION 'you''re not allowed to rewrite the table %',
                        table_oid::regclass;
  END IF;

  SELECT INTO pages relpages FROM pg_class WHERE oid = table_oid;
  IF pages > max_pages
  THEN
        RAISE EXCEPTION 'rewrites only allowed for table with less than % pages',
                        max_pages;
  END IF;

  IF current_hour NOT BETWEEN 1 AND 6
  THEN
        RAISE EXCEPTION 'rewrites only allowed between 1am and 6am';
  END IF;
END;
$$;

CREATE EVENT TRIGGER no_rewrite_allowed
                  ON table_rewrite
   EXECUTE PROCEDURE no_rewrite();
```

Глава 38. Система правил

В этой главе обсуждается система правил, реализованная в Postgres Pro. Промышленные системы правил по сути довольно простые, но при их использовании приходится сталкиваться с множеством неочевидных вещей.

В некоторых других базах данных определяются активные правила баз данных, которые обычно реализуются в виде процедур и триггеров. Так же их можно реализовать и в Postgres Pro.

Система правил (точнее говоря, система правил перезаписи запросов) полностью отличается от механизма хранимых процедур и триггеров. Она изменяет запросы по заданным правилам, а затем передаёт модифицированный запрос планировщику для планирования и выполнения. Это очень мощное средство, подходящее для решения множества задач, например, для определения представлений и процедур на языке запросов или реализации версионности. Теоретические основы и преимущества этой системы правил также описаны в ston90b и ong90 (на английском языке).

38.1. Дерево запроса

Чтобы понять, как работает система правил, нужно знать, когда она вызывается, что принимает на вход и какой результат выдаёт.

Система правил внедрена между анализатором запросов и планировщиком. Она принимает разобранный запрос, одно дерево запроса, и определённые пользователем правила перезаписи, тоже представленные деревьями с некоторой дополнительной информацией, и создаёт некоторое количество деревьев запросов в результате. Таким образом, на входе и выходе этой системы оказывается то, что может сформировать анализатор запросов, и как следствие, всё, с чем работает эта система, представимо в виде операторов SQL.

Так что же такое дерево запроса? Это внутреннее представление оператора SQL, в котором все образующие его части хранятся отдельно. Эти деревья можно увидеть в журнале сервера, если установить параметры конфигурации `debug_print_parse`, `debug_print_rewritten` или `debug_print_plan`. Действия правил также хранятся в виде деревьев запросов, в системном каталоге `pg_rewrite`. Они не форматируются как при выводе в журнал, но содержат точно такую же информацию.

Для прочтения неформатированного дерева требуется некоторый навык. Но так как представления дерева запросов в виде SQL достаточно, чтобы понять систему правил, в этой главе не будет рассказываться, как их читать.

Читая SQL-представления деревьев запросов в этой главе, необходимо понимать, на какие части разбивается оператор, когда он преобразуется в структуру дерева запроса. Дерево запроса состоит из следующих частей:

тип команды

Это простое значение, говорящее, какая команда (`SELECT`, `INSERT`, `UPDATE` или `DELETE`) сгенерировала дерево запросов.

список отношений

Список отношений представляет собой массив отношений, используемых в запросе. В запросе `SELECT` он включает отношения, указанные после ключевого слова `FROM`.

Каждый элемент списка отношений представляет таблицу или представление и говорит, с каким именем они упоминаются в других частях запроса. В дереве запросов записываются номера элементов списка отношений, а не их имена, поэтому для него неактуальна проблема дублирования имён, как для оператора SQL. Такая проблема может возникнуть при объединении списков отношений, образованных разными правилами. В этой главе данная ситуация рассматриваться не будет.

результирующее отношение

Индекс в списке отношений, указывающий на отношение, которое будет получать результаты запроса.

В запросах SELECT результирующее отношение отсутствует. (Особый случай SELECT INTO практически равнозначен CREATE TABLE с последующим INSERT ... SELECT и здесь отдельно не рассматривается.)

Для команд INSERT, UPDATE и DELETE результирующим отношением будет таблица (или представление!), в которой будут происходить изменения.

выходной список

Выходной список — это список выражений, определяющих результат запроса. В случае SELECT, это выражения, которые образуют окончательный набор выходных данных. Они соответствуют выражениям, записанным между ключевыми словами SELECT и FROM. (Указание * — это просто краткое обозначение имён всех столбцов отношения. Анализатор разворачивает его в список отдельных столбцов, так что система правил никогда не видит его.)

Командам DELETE не нужен обычный выходной список, так как они не выдают никакие результаты. Вместо этого планировщик добавляет в пустой выходной список специальную запись CTID, чтобы исполнитель мог найти удаляемую строку. (CTID добавляется, когда результирующее отношение — обычная таблица. Если это представление, планировщиком добавляется переменная, содержащая всю строку, как рассказывается в Подразделе 38.2.4.)

Для команд INSERT выходной список описывает новые строки, которые должны попасть в результирующее отношение. Он включает выражения в предложении VALUES или предложении SELECT в INSERT ... SELECT. На первом этапе процесс перезаписи добавляет элементы выходного списка для столбцов, которым ничего не присвоила исходная команда, но имеющих значения по умолчанию. Все остальные столбцы (без заданного значения и значения по умолчанию) планировщик заполняет константой NULL.

Для команд UPDATE выходной список описывает новые строки, которые должны заменить старые. В системе правил он содержит только выражения из части SET столбец = выражение. Для пропущенных столбцов планировщик вставляет выражения, копирующие значения из старой строки в новую. Так же, как и с командой DELETE, при этом добавляется CTID или переменная со всей строкой, чтобы исполнитель мог найти изменяемую старую строку.

Каждая запись в выходном списке содержит выражение, которое может быть константой, переменной, указывающей на столбец отношения в таблице отношений, параметром или деревом выражений, образованным из констант, переменных, операторов, вызовов функций и т. д.

условие фильтра

Условие фильтра запроса — это выражение, во многом похожее на те, что содержатся в выходном списке. Результат этого выражения — логический, он говорит, должна ли выполняться операция (INSERT, UPDATE, DELETE или SELECT) для данной строки в результате. Оно соответствует предложению WHERE SQL-оператора.

дерево соединения

Дерево соединения запроса показывает структуру предложения FROM. Для простых запросов вида SELECT ... FROM a, b, c, дерево соединения — это просто список элементов FROM, так как они могут соединяться в любом порядке. Но с выражениями JOIN, особенно с внешними соединениями, приходится соединять отношения именно в заданном порядке. В этом случае дерево соединения отражает структуру выражений JOIN. Ограничения, связанные с конкретными предложениями JOIN (из выражений ON или USING), тоже сохраняются в виде условных выражений, добавленных к соответствующим узлам дерева соединения. Как оказалось, выражение WHERE верхнего уровня тоже удобно хранить как условие, добавленное к элементу верхнего уровня дерева соединения. Поэтому в дереве соединения на самом деле представляются оба предложения оператора SELECT — FROM и WHERE.

другие

Другие части дерева запроса, например, предложение ORDER BY, в данном контексте не представляют интереса. Система правил выполняет в них некоторые подстановки, применяя правила, но это не имеет непосредственного отношения к основам системы правил.

38.2. Система правил и представления

Представления в Postgres Pro реализованы на основе системы правил. Фактически по сути нет никакого отличия

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

от следующих двух команд:

```
CREATE TABLE myview (same column list as mytab);
CREATE RULE "_RETURN" AS ON SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

так как именно эти действия CREATE VIEW выполняет внутри. Это имеет некоторые побочные эффекты. В частности, информация о представлениях в системных каталогах Postgres Pro ничем не отличается от информации о таблицах. Поэтому при анализе запроса нет абсолютно никакой разницы между таблицами и представлениями. Они представляют собой одно и то же — отношения.

38.2.1. Как работают правила SELECT

Правила ON SELECT применяются ко всем запросам на последнем этапе, даже если это команда INSERT, UPDATE или DELETE. Эти правила отличаются от правил других видов тем, что они модифицируют непосредственно дерево запросов, а не создают новое. Поэтому мы начнём описание с правил SELECT.

В настоящее время возможно только одно действие в правиле ON SELECT и это должно быть безусловное действие SELECT, выполняемое в режиме INSTEAD. Это ограничение было введено, чтобы сделать правила достаточно безопасными для применения обычными пользователями, так что действие правил ON SELECT сводится к реализации представлений.

В примерах этой главы рассматриваются два представления с соединением, которые выполняют некоторые вычисления, и которые, в свою очередь, используются другими представлениями. Первое из этих двух представлений затем модифицируется, к нему добавляются правила для операций INSERT, UPDATE и DELETE, так что в итоге получается представление, которое работает как обычная таблица с некоторыми необычными функциями. Это не самый простой пример для начала, поэтому понять некоторые вещи будет сложнее. Но лучше иметь один пример, поэтапно охватывающий все обсуждаемые здесь темы, чем несколько различных, при восприятии которых в итоге может возникнуть путаница.

Таблицы, которые понадобятся нам для описания системы правил, выглядят так:

```
CREATE TABLE shoe_data (
    shoename   text,          -- первичный ключ
    sh_avail   integer,       -- число имеющихся пар
    slcolor    text,          -- предпочитаемый цвет шнурков
    slminlen   real,          -- минимальная длина шнурков
    slmaxlen   real,          -- максимальная длина шнурков
    slunit     text           -- единица длины
);

CREATE TABLE shoelace_data (
    sl_name    text,          -- первичный ключ
    sl_avail   integer,       -- число имеющихся пар
    sl_color   text,          -- цвет шнурков
    sl_len     real,          -- длина шнурков
```

```
    sl_unit    text           -- единица длины
);

CREATE TABLE unit (
    un_name    text,          -- первичный ключ
    un_fact    real           -- коэффициент для перевода в см
);
```

Как можно догадаться, в них хранятся данные обувной фабрики.

Представления создаются так:

```
CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
           sh.slcolor,
           sh.slminlen,
           sh.slminlen * un.un_fact AS slminlen_cm,
           sh.slmaxlen,
           sh.slmaxlen * un.un_fact AS slmaxlen_cm,
           sh.slunit
      FROM shoe_data sh, unit un
     WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
    SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
           s.sl_len,
           s.sl_unit,
           s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
     WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           least(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM shoe rsh, shoelace rsl
     WHERE rsl.sl_color = rsh.slcolor
       AND rsl.sl_len_cm >= rsh.slminlen_cm
       AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

Команда `CREATE VIEW` для представления `shoelace` (самого простого из имеющихся) создаёт отношение `shoelace` и запись в `pg_rewrite` о правиле перезаписи, которое должно применяться, когда в запросе на выборку задействуется отношение `shoelace`. Для этого правила не задаются условия применения (о них рассказывается ниже, в описании правил не для `SELECT`, так как правила `SELECT` в настоящее бывают только безусловными) и оно действует в режиме `INSTEAD`. Заметьте, что условия применения отличаются от условий фильтра запроса, например, действие для нашего правила содержит условие фильтра. Действие правила выражается одним деревом запроса, которое является копией оператора `SELECT` в команде, создающей представление.

> **Примечание**
>
> Два дополнительных элемента списка отношений `NEW` и `OLD`, которые можно увидеть в соответствующей строке `pg_rewrite`, не представляют интереса для правил `SELECT`.

Сейчас мы наполним таблицы `unit` (единицы измерения), `shoe_data` (данные о туфлях) и `shoelace_data` (данные о шнурках) и выполним простой запрос к представлению:

```
INSERT INTO unit VALUES ('cm', 1.0);
INSERT INTO unit VALUES ('m', 100.0);
INSERT INTO unit VALUES ('inch', 2.54);

INSERT INTO shoe_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO shoe_data VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO shoe_data VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO shoe_data VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');

INSERT INTO shoelace_data VALUES ('sl1', 5, 'black', 80.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl3', 0, 'black', 35.0, 'inch');
INSERT INTO shoelace_data VALUES ('sl4', 8, 'black', 40.0, 'inch');
INSERT INTO shoelace_data VALUES ('sl5', 4, 'brown', 1.0, 'm');
INSERT INTO shoelace_data VALUES ('sl6', 0, 'brown', 0.9, 'm');
INSERT INTO shoelace_data VALUES ('sl7', 7, 'brown', 60, 'cm');
INSERT INTO shoelace_data VALUES ('sl8', 1, 'brown', 40, 'inch');

SELECT * FROM shoelace;

 sl_name   | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
-----------+----------+----------+--------+---------+-----------
 sl1       |        5 | black    |     80 | cm      |        80
 sl2       |        6 | black    |    100 | cm      |       100
 sl7       |        7 | brown    |     60 | cm      |        60
 sl3       |        0 | black    |     35 | inch    |      88.9
 sl4       |        8 | black    |     40 | inch    |     101.6
 sl8       |        1 | brown    |     40 | inch    |     101.6
 sl5       |        4 | brown    |      1 | m       |       100
 sl6       |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

Это самый простой запрос `SELECT`, который можно выполнить с нашими представлениями, и мы воспользуемся этим, чтобы объяснить азы правил представлений. Запрос `SELECT * FROM shoelace` интерпретируется анализатором запросов и преобразуется в дерево запроса:

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace;
```

Это дерево передаётся в систему правил, которая проходит по списку отношений и проверяет, есть ли какие-либо правила для этих отношений. Обрабатывая элемент отношения `shoelace` (сейчас он единственный), система правил находит правило `_RETURN` с деревом запроса:

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       s.sl_len * u.un_fact AS sl_len_cm
  FROM shoelace old, shoelace new,
       shoelace_data s, unit u
 WHERE s.sl_unit = u.un_name;
```

Чтобы развернуть представление, механизм перезаписи просто формирует новый элемент для списка отношений — подзапрос, содержащий дерево действия правила, и подставляет этот элемент вместо исходного, на который ссылалось представление. Получившееся перезаписанное дерево запроса будет почти таким как дерево запроса:

```
SELECT shoelace.sl_name, shoelace.sl_avail,
```

```
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM (SELECT s.sl_name,
               s.sl_avail,
               s.sl_color,
               s.sl_len,
               s.sl_unit,
               s.sl_len * u.un_fact AS sl_len_cm
          FROM shoelace_data s, unit u
         WHERE s.sl_unit = u.un_name) shoelace;
```

Однако есть одно различие: в списке отношений подзапроса будут содержаться два дополнительных элемента: `shoelace old` и `shoelace new`. Эти элементы не принимают непосредственного участия в запросе, так как они не задействованы в дереве соединения подзапроса и в целевом списке. Механизм перезаписи использует их для хранения информации о проверке прав доступа, которая изначально хранилась в элементе, указывающем на представление. Таким образом, исполнитель будет по-прежнему проверять, имеет ли пользователь необходимые права для доступа к представлению, хотя в перезаписанном запросе это представление не фигурирует непосредственно.

Так было применено первое правило. Система правил продолжит проверку оставшихся элементов списка отношений на верхнем уровне запроса (в данном случае таких элементов нет) и рекурсивно проверит элементы списка отношений в добавленном подзапросе, не ссылаются ли они на представления. (Но `old` и `new` разворачиваться не будут — иначе мы получили бы бесконечную рекурсию!) В этом примере для `shoelace_data` и `unit` нет правил перезаписи, так что перезапись завершается и результат, полученный выше, передаётся планировщику.

Сейчас мы хотим написать запрос, который выбирает туфли из имеющихся в данный момент, для которых есть подходящие шнурки (по цвету и длине) и число готовых пар больше или равно двум.

```
SELECT * FROM shoe_ready WHERE total_avail >= 2;

 shoename | sh_avail | sl_name | sl_avail | total_avail
----------+----------+---------+----------+-------------
 sh1      |        2 | sl1     |        5 |           2
 sh3      |        4 | sl7     |        7 |           4
(2 rows)
```

На этот раз анализатор запроса выводит такое дерево:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM shoe_ready shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

Первое правило применяется к представлению `shoe_ready` и в результате получается дерево запроса:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               least(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM shoe rsh, shoelace rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
```

```
 WHERE shoe_ready.total_avail >= 2;
```

Подобным образом, правила для `shoe` и `shoelace` подставляются в список отношений, что даёт окончательное дерево запроса:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               least(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM (SELECT sh.shoename,
                       sh.sh_avail,
                       sh.slcolor,
                       sh.slminlen,
                       sh.slminlen * un.un_fact AS slminlen_cm,
                       sh.slmaxlen,
                       sh.slmaxlen * un.un_fact AS slmaxlen_cm,
                       sh.slunit
                  FROM shoe_data sh, unit un
                 WHERE sh.slunit = un.un_name) rsh,
               (SELECT s.sl_name,
                       s.sl_avail,
                       s.sl_color,
                       s.sl_len,
                       s.sl_unit,
                       s.sl_len * u.un_fact AS sl_len_cm
                  FROM shoelace_data s, unit u
                 WHERE s.sl_unit = u.un_name) rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail > 2;
```

Это может показаться неэффективным, но планировщик преобразует этот запрос в одноуровневое дерево, «подтягивая» подзапросы в главный запрос, а затем планирует соединения так же, как и при явной записи с соединениями. Таким образом, упрощение дерева запросов является оптимизацией, которая производится независимо от перезаписи запросов.

38.2.2. Правила представлений не для `SELECT`

До этого в описании правил представлений не затрагивались два компонента дерева запросов — тип команды и результирующее отношение. На самом деле, тип команды не важен для правил представления, но результирующее отношение может повлиять на работу механизма перезаписи, потому что если это представление, требуются дополнительные операции.

Есть только несколько отличий между деревом запроса для `SELECT` и деревом для другой команды. Очевидно, у них различные типы команд, и для команды, отличной от `SELECT`, результирующее отношение указывает на элемент в списке отношений, куда должен попасть результат. Все остальные компоненты в точности те же. Поэтому, например, если взять таблицы `t1` и `t2` со столбцами `a` и `b`, деревья запросов для этих операторов:

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;
```

будут практически одинаковыми. В частности:

- Списки отношений содержат элементы для таблиц `t1` и `t2`.

- Выходные списки содержат одну переменную, указывающую на столбец b элемента-отношения для таблицы t2.
- Выражения условий сравнивают столбцы a обоих элементов-отношений на равенство.
- Деревья соединений показывают простое соединение между t1 и t2.

Как следствие, для обоих деревьев строятся похожие планы выполнения, с соединением двух таблиц. Для UPDATE планировщик добавляет в выходной список недостающие столбцы из t1 и окончательное дерево становится таким:

```
UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;
```

В результате исполнитель, обрабатывающий соединение, выдаёт тот же результат, что и запрос:

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

Но с UPDATE есть маленькая проблема: часть плана исполнителя, в которой выполняется соединение, не представляет, для чего предназначены результаты соединения. Она просто выдаёт результирующий набор строк. Фактически есть одна команда SELECT, а другая, UPDATE, обрабатывается исполнителем выше, где он уже знает, что это команда UPDATE и что результат должен попасть в таблицу t1. Но какие из строк таблицы должны заменяться новыми?

Для решения этой проблемы в выходной список операторов UPDATE (и DELETE) добавляется ещё один элемент: идентификатор текущего кортежа (Current Tuple ID, CTID). Это системный столбец, содержащий номер блока в файле и позицию строки в блоке. Зная таблицу, по CTID можно получить исходную строку в t1, подлежащую изменению. С добавленным в выходной список CTID запрос фактически выглядит так:

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

Теперь мы перейдём ещё к одной особенности Postgres Pro. Старые строки таблицы не переписываются, поэтому ROLLBACK выполняется быстро. С командой UPDATE в таблицу вставляется новая строка результата (без CTID) и в заголовке старой строки, на которую указывает CTID, в поля cmax и xmax записываются текущий счётчик команд и идентификатор текущей транзакции. Таким образом, старая строка оказывается скрытой и после фиксирования транзакции процесс очистки может окончательно удалить неактуальную версию строки.

Зная всё это, мы можем применять правила представлений абсолютно таким же образом к любой команде — никаких различий нет.

38.2.3. Преимущества представлений в Postgres Pro

Выше было показано, как система правил внедряет определения представлений в исходное дерево запроса. Во втором примере простой запрос SELECT к одному представлению создал окончательное дерево запроса, соединяющее 4 таблицы (таблица unit использовалась дважды с разными именами).

Преимущество реализации представлений через систему правил заключается в том, что планировщик получает в одном дереве запроса всю информацию о таблицах, которые нужно прочитать, о том, как связаны эти таблицы, об условиях в представлениях, а также об условиях, заданных в исходном запросе. И всё это имеет место, когда сам исходный запрос представляет собой соединение представлений. Планировщик должен выбрать лучший способ выполнения запроса, и чем больше информации он получит, тем лучше может быть его выбор. И то, как в Postgres Pro реализована система правил, гарантирует, что ему поступает вся информация, собранная о запросе на данный момент.

38.2.4. Изменение представления

Но что произойдёт, если записать имя представления в качестве целевого отношения команды INSERT, UPDATE или DELETE? Если проделать подстановки, описанные выше, будет получено дерево запроса, в котором результирующее отношение указывает на элемент-подзапрос, что не будет работать. Однако Postgres Pro даёт ряд возможностей, чтобы сделать представления изменяемыми.

Если подзапрос выбирает данные из одного базового отношения и он достаточно прост, механизм перезаписи может автоматически заменить его нижележащим базовым отношением, чтобы команды INSERT, UPDATE или DELETE обращались к базовому отношению. Представления, «достаточно простые» для этого, называются *автоматически изменяемыми*. Подробнее виды представлений, которые могут изменяться автоматически, описаны в CREATE VIEW.

Эту задачу также можно решить, создав триггер INSTEAD OF для представления. В этом случае перезапись будет работать немного по-другому. Для INSERT механизм перезаписи не делает с представлением ничего, оставляя его результирующим отношением запроса. Для UPDATE и DELETE ему по-прежнему придётся разворачивать запрос представления, чтобы получить «старые» строки, которые эта команда попытается изменить или удалить. Поэтому представление разворачивается как обычно, но в запрос добавляется ещё один элемент списка отношений, указывающий на представление в роли результирующего отношения.

При этом возникает проблема идентификации строк в представлении, подлежащих изменению. Вспомните, что когда результирующее отношение является таблицей, в выходной список добавляется специальное поле CTID, указывающее на физическое расположение изменяемых строк. Но это не будет работать, когда результирующее отношение — представление, так как в представлениях нет CTID, потому что их строки физически нигде не находятся. Вместо этого, для операций UPDATE или DELETE в выходной список добавляется специальный элемент wholerow (вся строка), который разворачивается в содержимое всех столбцов представления. Используя этот элемент, исполнитель передаёт строку «old» в триггер INSTEAD OF. Какие именно строки должны изменяться фактически, будет решать сам триггер, исходя из полученных значений старых и новых строк.

Кроме того, пользователь может определить правила INSTEAD, в которых задать действия замены для команд INSERT, UPDATE и DELETE с представлением. Эти правила обычно преобразуют команду в другую команду, изменяющую одну или несколько таблиц, а не представление. Эта тема освещается в Разделе 38.4.

Заметьте, что такие правила вычисляются сначала, перезаписывая исходный запрос до того, как он будет планироваться и выполняться. Поэтому, если для представления определены и триггеры INSTEAD OF, и правила для INSERT, UPDATE или DELETE, сначала вычисляются правила, а в зависимости от их действия, триггеры могут не вызываться вовсе.

Автоматическая перезапись запросов INSERT, UPDATE или DELETE с простыми представлениями всегда производится в последнюю очередь. Таким образом, если у представления есть правила или триггеры, они переопределяют поведение автоматически изменяемых представлений.

Если для представления не определены правила INSTEAD или триггеры INSTEAD OF, и запрос не удаётся автоматически переписать в виде обращения к нижележащему базовому отношению, возникает ошибка, потому что исполнитель не сможет изменить такое представление.

38.3. Материализованные представления

Материализованные представления в Postgres Pro основаны на системе правил, как и представления, но их содержимое сохраняется как таблица. Основное отличие между:

```
CREATE MATERIALIZED VIEW mymatview AS SELECT * FROM mytab;
```

и этой командой:

```
CREATE TABLE mymatview AS SELECT * FROM mytab;
```

состоит в том, что материализованное представление впоследствии нельзя будет изменить непосредственно, а запрос, создающий материализованное представление, сохраняется точно так же, как запрос представления, и получить актуальные данные в материализованном представлении можно так:

```
REFRESH MATERIALIZED VIEW mymatview;
```

Информация о материализованном представлении в системных каталогах Postgres Pro ничем не отличается от информации о таблице или представлении. Поэтому для анализатора

запроса материализованное представление является просто отношением, как таблица или представление. Когда запрос обращается к материализованному представлению, данные возвращаются непосредственно из него, как из таблицы; правило применяется, только чтобы его наполнить.

Хотя обращение к данным в материализованном представлении часто выполняется гораздо быстрее, чем обращение к нижележащим таблицам напрямую или через представление, данные в нём не всегда актуальные (но иногда это вполне приемлемо). Рассмотрим таблицу с данными продаж:

```
CREATE TABLE invoice (
    invoice_no    integer        PRIMARY KEY,
    seller_no     integer,       -- идентификатор продавца
    invoice_date  date,          -- дата продажи
    invoice_amt   numeric(13,2)  -- сумма продажи
);
```

Если пользователям нужно быстро обработать исторические данные, возможно их интересуют только общие показатели, а полнота данных на текущий момент не важна:

```
CREATE MATERIALIZED VIEW sales_summary AS
  SELECT
      seller_no,
      invoice_date,
      sum(invoice_amt)::numeric(13,2) as sales_amt
    FROM invoice
    WHERE invoice_date < CURRENT_DATE
    GROUP BY
      seller_no,
      invoice_date;

CREATE UNIQUE INDEX sales_summary_seller
  ON sales_summary (seller_no, invoice_date);
```

Это материализованное представление может быть полезно для построения графика в информационной панели менеджеров по продажам. Для ежесуточного обновления статистики можно запланировать задание по расписанию, которое будет выполнять этот оператор:

```
REFRESH MATERIALIZED VIEW sales_summary;
```

Ещё одно применение материализованного представления — предоставить быстрый доступ к данным, получаемым с удалённой системы через обёртку сторонних данных. Ниже приведён простой пример с обёрткой `file_fdw`, с замерами времени, но так как при этом использовался кеш локальной системы, выигрыш в производительности при обращении к удалённой системе обычно будет гораздо больше, чем показано здесь. Заметьте, что мы также использовали возможность добавить индекс в материализованное представление, тогда как `file_fdw` индексы не поддерживает; при других видах доступа к сторонним данным такого преимущества может не быть.

Подготовка:

```
CREATE EXTENSION file_fdw;
CREATE SERVER local_file FOREIGN DATA WRAPPER file_fdw;
CREATE FOREIGN TABLE words (word text NOT NULL)
  SERVER local_file
  OPTIONS (filename '/usr/share/dict/words');
CREATE MATERIALIZED VIEW wrd AS SELECT * FROM words;
CREATE UNIQUE INDEX wrd_word ON wrd (word);
CREATE EXTENSION pg_trgm;
CREATE INDEX wrd_trgm ON wrd USING gist (word gist_trgm_ops);
VACUUM ANALYZE wrd;
```

Теперь давайте проверим написание слова. Сначала непосредственно через обёртку `file_fdw`:

```
SELECT count(*) FROM words WHERE word = 'caterpiler';

 count
-------
     0
(1 row)
```

Выполнив `EXPLAIN ANALYZE`, мы получаем:

```
 Aggregate  (cost=21763.99..21764.00 rows=1 width=0) (actual time=188.180..188.181
 rows=1 loops=1)
   ->  Foreign Scan on words  (cost=0.00..21761.41 rows=1032 width=0) (actual
 time=188.177..188.177 rows=0 loops=1)
         Filter: (word = 'caterpiler'::text)
         Rows Removed by Filter: 479829
         Foreign File: /usr/share/dict/words
         Foreign File Size: 4953699
 Planning time: 0.118 ms
 Execution time: 188.273 ms
```

Если же теперь обратиться к материализованному представлению, запрос выполнится гораздо быстрее:

```
 Aggregate  (cost=4.44..4.45 rows=1 width=0) (actual time=0.042..0.042 rows=1 loops=1)
   ->  Index Only Scan using wrd_word on wrd  (cost=0.42..4.44 rows=1 width=0) (actual
 time=0.039..0.039 rows=0 loops=1)
         Index Cond: (word = 'caterpiler'::text)
         Heap Fetches: 0
 Planning time: 0.164 ms
 Execution time: 0.117 ms
```

В любом случае слово записано неправильно, поэтому давайте попробуем найти то, что имелось в виду. Сначала опять через `file_fdw`:

```
SELECT word FROM words ORDER BY word <-> 'caterpiler' LIMIT 10;

     word
---------------
 cater
 caterpillar
 Caterpillar
 caterpillars
 caterpillar's
 Caterpillar's
 caterer
 caterer's
 caters
 catered
(10 rows)
```

```
 Limit  (cost=11583.61..11583.64 rows=10 width=32) (actual time=1431.591..1431.594
 rows=10 loops=1)
   ->  Sort  (cost=11583.61..11804.76 rows=88459 width=32) (actual
 time=1431.589..1431.591 rows=10 loops=1)
         Sort Key: ((word <-> 'caterpiler'::text))
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Foreign Scan on words  (cost=0.00..9672.05 rows=88459 width=32) (actual
 time=0.057..1286.455 rows=479829 loops=1)
               Foreign File: /usr/share/dict/words
               Foreign File Size: 4953699
```

```
Planning time: 0.128 ms
Execution time: 1431.679 ms
```

Затем через материализованное представление:

```
Limit  (cost=0.29..1.06 rows=10 width=10) (actual time=187.222..188.257 rows=10
loops=1)
  ->  Index Scan using wrd_trgm on wrd  (cost=0.29..37020.87 rows=479829 width=10)
(actual time=187.219..188.252 rows=10 loops=1)
        Order By: (word <-> 'caterpiler'::text)
Planning time: 0.196 ms
Execution time: 198.640 ms
```

Если периодическое обновление данных из другого источника в локальной базе данных вас устраивает, этот подход может дать значительный выигрыш в скорости.

38.4. Правила для INSERT, UPDATE и DELETE

Правила, определяемые для команд INSERT, UPDATE и DELETE, значительно отличаются от правил представлений, описанных в предыдущем разделе. Во-первых, команда CREATE RULE позволяет создавать правила со следующими особенностями:

- Они могут не определять действия.
- Они могут определять несколько действий.
- Они могут действовать в режиме INSTEAD или ALSO (по умолчанию).
- Становятся полезными псевдоотношения NEW и OLD.
- Они могут иметь условия применения.

Во-вторых, они не модифицируют само исходное дерево запроса. Вместо этого они создают несколько новых деревьев запросов и могут заменить исходное.

> **Внимание**
>
> Во многих случаях для задач, выполнимых с использованием правил для INSERT/UPDATE/DELETE, лучше применять триггеры. Оформляются триггеры чуть сложнее, но понять их смысл гораздо проще. К тому же с правилами могут быть получены неожиданные результаты, когда исходный запрос содержит изменчивые функции: в процессе исполнения правил эти функции могут вызываться большее число раз, чем ожидается.
>
> Кроме того, в некоторых случаях эти типы правил вообще нельзя применять; а именно, с предложениями WITH в исходном запросе и с вложенными подзапросами SELECT с множественным присваиванием в списке SET запросов UPDATE. Это объясняется тем, что копирование этих конструкций в запрос правила привело бы к многократному вычислению вложенного запроса, что пошло бы в разрез с выраженными намерениями автора запроса.

38.4.1. Как работают правила для изменения

Запомните синтаксис:

```
CREATE [ OR REPLACE ] RULE имя AS ON событие
    TO таблица [ WHERE условие ]
    DO [ ALSO | INSTEAD ] { NOTHING | команда | ( команда ; команда ... ) }
```

В дальнейшем, под *правилами для изменения* подразумеваются правила, определяемые для команд INSERT, UPDATE или DELETE.

Правила для изменения применяются системой правил, когда результирующее отношение и тип команды в дереве запроса совпадает с объектом и событием, заданным в команде CREATE RULE. Для такого правила система правил создаёт список деревьев запросов. Изначально этот список пуст.

С правилом может быть связано ноль (ключевое слово NOTHING), одно или несколько действий. Простоты ради мы рассмотрим правило с одним действием. Правило может иметь, а может не иметь условия применения, и действует в режиме INSTEAD или ALSO (по умолчанию).

Что такое условие применения правила? Это условие, которое говорит, когда нужно, а когда не нужно применять действия правила. В этом условии можно обращаться к псевдоотношениям NEW и/или OLD, которые представляют целевое отношение (но с особым значением).

Всего есть три варианта формирования деревьев запросов для правила с одним действием.

Без условия применения в режиме ALSO или INSTEAD

дерево запроса из действия правила с добавленным условием исходного дерева

С условием применения в режиме ALSO

дерево запроса из действия правила с условием применения правила и условием, добавленным из исходного дерева

С условием применения в режиме INSTEAD

дерево запроса из действия правила с условием применения правила и условием из исходного дерева; также добавляется исходное дерево запроса с условием, обратным условию применения правила

Наконец, для правил ALSO в список добавляется исходное дерево запроса без изменений. Так как исходное дерево запроса также добавляют только правила INSTEAD с условиями применения, в итоге для правила с одним действием мы можем получить только одно или два дерева запросов.

Для правил ON INSERT исходный запрос (если он не перекрывается режимом INSTEAD) выполняется перед действиями, добавленными правилами. Поэтому эти действия могут видеть вставленные строки. Но для правил ON UPDATE и ON DELETE исходный запрос выполняется после действий, добавленных правилами. При таком порядке эти действия будут видеть строки, подлежащие изменению или удалению; иначе бы действия не работали, не найдя строк, соответствующих их условиям применения (эти строки уже будут изменены или удалены).

Деревья запросов, полученные из действий правил, снова попадают в систему перезаписи, где могут примениться дополнительные правила, добавляющие или убирающие деревья запроса. Поэтому действия правила должны выполнять команды другого типа или работать с другим результирующим отношением, иначе возникнет бесконечная рекурсия. (Система выявляет подобное рекурсивное разворачивание правил и выдаёт ошибку.)

Деревья запросов, заданные для действий в системном каталоге pg_rewrite, представляют собой только шаблоны. Так как они могут обращаться к элементам NEW и OLD в списке отношений, их можно будет использовать только после некоторых подстановок. В случае ссылки на NEW соответствующий элемент ищется в целевом списке исходного запроса. Если он найден, ссылка заменяется выражением этого элемента. В противном случае NEW означает то же самое, что и OLD (для команды UPDATE) или заменяется значением NULL (для команды INSERT). Любые ссылки на OLD заменяются ссылкой на элемент результирующего отношения в списке отношений.

После того как система применит все правила для изменения, она применяет правила представления к полученному дереву (или деревьям) запроса. Представления не могут добавлять новые действия для изменения, поэтому нет необходимости применять такие правила к результату перезаписи представления.

38.4.1.1. Пошаговый разбор первого правила

Предположим, что нам нужно отслеживать изменения в столбце sl_avail таблицы shoelace_data. Мы можем создать таблицу для ведения журнала и правило, которое будет добавлять в неё записи по условию, когда для shoelace_data выполняется UPDATE.

```
CREATE TABLE shoelace_log (
```

```
    sl_name    text,          -- шнурки, количество которых изменилось
    sl_avail   integer,       -- новое количество
    log_who    text,          -- кто изменил
    log_when   timestamp      -- когда
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail <> OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
                                    NEW.sl_name,
                                    NEW.sl_avail,
                                    current_user,
                                    current_timestamp
                                );
```

Теперь, если кто-то выполнит:

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

мы увидим в таблице журнала:

```
SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who | log_when
---------+----------+---------+----------------------------------
 sl7     |        6 | Al      | Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

Именно это нам и нужно. При этом внутри происходит следующее. Анализатор запроса создаёт дерево:

```
UPDATE shoelace_data SET sl_avail = 6
  FROM shoelace_data shoelace_data
 WHERE shoelace_data.sl_name = 'sl7';
```

В системном каталоге находится правило `log_shoelace`, настроенное на изменение (`ON UPDATE`) с условием применения:

```
NEW.sl_avail <> OLD.sl_avail
```

и действием:

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old;
```

(Это выглядит несколько странно, так как обычно нельзя написать `INSERT ... VALUES ... FROM`. Предложение `FROM` здесь добавлено, просто чтобы показать, что в дереве запроса для ссылок `new` и `old` есть элементы в списке отношений. Они необходимы для того, чтобы к ним могли обращаться переменные в дереве запроса команды `INSERT`.)

Так как это правило `ALSO` с условием применения, система правил должна выдать два дерева запросов: изменённое действие правила и исходное дерево запроса. На первом шаге список отношений исходного запроса вставляется в дерево действия правила и получается:

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data;
```

На втором шаге в это дерево добавляется условие применения правила, так что результирующий набор ограничивается строками, в которых меняется `sl_avail`:

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE new.sl_avail <> old.sl_avail;
```

(Это выглядит ещё более странно, ведь в `INSERT ... VALUES` не записывается и предложение `WHERE`, но планировщик и исполнитель не испытывают затруднений с этим. Они всё равно должны поддерживать эту функциональность для `INSERT ... SELECT`.)

На третьем шаге добавляется условие исходного дерева, что ещё больше ограничивает результирующий набор, оставляя в нём только строки, которые затронул бы исходный запрос:

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE new.sl_avail <> old.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

На четвёртом шаге ссылки на `NEW` заменяются элементами выходного списка из исходного дерева запроса или переменными из результирующего отношения:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE 6 <> old.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

На последнем, пятом шаге ссылки на `OLD` заменяются ссылками на результирующее отношение:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

Вот и всё. Так как правило действует в режиме `ALSO`, мы также выводим исходное дерево запроса. Таким образом, система правил выдаёт список с двумя деревьями запросов, соответствующими этим операторам:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
 WHERE sl_name = 'sl7';
```

Они выполняются в показанном порядке и именно это должно делать данное правило.

Благодаря заменам и добавленным условиям в журнал не добавится запись, например, при таком исходном запросе:

```
UPDATE shoelace_data SET sl_color = 'green'
 WHERE sl_name = 'sl7';
```

В этом случае исходное дерево запроса не содержит элемент выходного списка для `sl_avail`, так что `NEW.sl_avail` будет заменено переменной `shoelace_data.sl_avail`. Таким образом, дополнительная команда, созданная правилом, будет такой:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, shoelace_data.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE shoelace_data.sl_avail <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

Это условие применения не будет выполняться никогда.

Это также будет работать, если исходный запрос изменяет несколько строк. Так, если кто-то выполнит команду:

```
UPDATE shoelace_data SET sl_avail = 0
 WHERE sl_color = 'black';
```

фактически будут изменены четыре строки (`sl1`, `sl2`, `sl3` и `sl4`). Но для `sl3` значение `sl_avail = 0`. В этом случае условие исходного дерева другое, так что это правило выдаёт такое дополнительное дерево запроса:

```
INSERT INTO shoelace_log
SELECT shoelace_data.sl_name, 0,
       current_user, current_timestamp
  FROM shoelace_data
 WHERE 0 <> shoelace_data.sl_avail
   AND shoelace_data.sl_color = 'black';
```

. С таким деревом запроса в журнал определённо будут добавлены три записи. И это абсолютно правильно.

Здесь мы видим, почему важно, чтобы исходное дерево запроса выполнялось в конце. Если бы оператор `UPDATE` выполнился сначала, все строки уже получили бы нулевые значения, так что записывающий в журнал `INSERT` не нашёл бы строк, в которых `0 <> shoelace_data.sl_avail`.

38.4.2. Сочетание с представлениями

Есть один простой вариант защититься от ранее упомянутой возможности выполнять `INSERT`, `UPDATE` или `DELETE` для представлений, когда это нежелательно — создать правила, просто отбрасывающие деревья этих запросов. В нашем случае они будут выглядеть так:

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
    DO INSTEAD NOTHING;
```

Если теперь кто-то попытается выполнить одну из этих операций с представлением `shoe`, система правил применит эти правила. Так как это правила без действий в режиме `INSTEAD`, результирующий список деревьев запроса будет пуст и весь запрос аннулируется, так что после работы системы правил будет нечего оптимизировать и выполнять.

Более сложный вариант — использовать систему правил для создания правил, преобразующих дерево запроса в выполняющее нужную операцию с реальными таблицами. Чтобы реализовать это с представлением `shoelace`, мы создадим следующие правила:

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
```

```
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    );

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
    DO INSTEAD
    UPDATE shoelace_data
       SET sl_name = NEW.sl_name,
           sl_avail = NEW.sl_avail,
           sl_color = NEW.sl_color,
           sl_len = NEW.sl_len,
           sl_unit = NEW.sl_unit
     WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
    DO INSTEAD
    DELETE FROM shoelace_data
     WHERE sl_name = OLD.sl_name;
```

Если вы хотите поддерживать также запросы к представлению с `RETURNING`, вам надо создать правила с предложениями `RETURNING`, которые будут вычислять строки представления. Это обычно довольно тривиально для представлений с одной нижележащей таблицей, но несколько затруднительно для представлений с соединением, таких как `shoelace`. Например, для `INSERT` это будет выглядеть так:

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    )
    RETURNING
           shoelace_data.*,
           (SELECT shoelace_data.sl_len * u.un_fact
            FROM unit u WHERE shoelace_data.sl_unit = u.un_name);
```

Заметьте, что это одно правило поддерживает запросы и `INSERT`, и `INSERT RETURNING` к этому представлению — предложение `RETURNING` просто игнорируется при обычном `INSERT`.

Теперь предположим, что на фабрику прибывает партия шнурков с объёмной сопроводительной накладной. Но вы не хотите вручную вносить по одной записи в представление `shoelace`. Вместо этого можно создать две маленькие таблицы: в первую вы будете вставлять записи из накладной, а вторая пригодится для специального приёма. Для этого мы выполним следующие команды:

```
CREATE TABLE shoelace_arrive (
    arr_name    text,
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     text,
    ok_quant    integer
```

```
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
    DO INSTEAD
    UPDATE shoelace
       SET sl_avail = sl_avail + NEW.ok_quant
     WHERE sl_name = NEW.ok_name;
```

Теперь вы можете наполнить таблицу `shoelace_arrive` данными о поступивших шнурках из накладной:

```
SELECT * FROM shoelace_arrive;

 arr_name | arr_quant
----------+-----------
 sl3      |        10
 sl6      |        20
 sl8      |        20
(3 rows)
```

Взгляните на текущие данные:

```
SELECT * FROM shoelace;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl3      |        0 | black    |     35 | inch    |      88.9
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl8      |        1 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

Теперь переместите прибывшие шнурки во вторую таблицу:

```
INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

Проверьте, что получилось:

```
SELECT * FROM shoelace ORDER BY sl_name;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl3      |       10 | black    |     35 | inch    |      88.9
 sl8      |       21 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |       20 | brown    |    0.9 | m       |        90
(8 rows)

SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who| log_when
---------+----------+--------+---------------------------------
 sl7     |        6 | Al     | Tue Oct 20 19:14:45 1998 MET DST
 sl3     |       10 | Al     | Tue Oct 20 19:25:16 1998 MET DST
```

```
 sl6     |       20 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl8     |       21 | Al     | Tue Oct 20 19:25:16 1998 MET DST
(4 rows)
```

Чтобы получить эти результаты из одного `INSERT ... SELECT`, была проделана большая работа. Мы подробно опишем всё преобразование дерева запросов в продолжении этой главы. Начнём с дерева, выданного анализатором запроса:

```
INSERT INTO shoelace_ok
SELECT shoelace_arrive.arr_name, shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

Теперь применяется первое правило `shoelace_ok_ins`, создающее такое дерево:

```
UPDATE shoelace
   SET sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace
 WHERE shoelace.sl_name = shoelace_arrive.arr_name;
```

и отбрасывающее исходный `INSERT` в `shoelace_ok`. Этот переписанный запрос снова поступает в систему правил и второе применяемое правило `shoelace_upd` выдаёт:

```
UPDATE shoelace_data
   SET sl_name = shoelace.sl_name,
       sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant,
       sl_color = shoelace.sl_color,
       sl_len = shoelace.sl_len,
       sl_unit = shoelace.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data
 WHERE shoelace.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = shoelace.sl_name;
```

Это тоже правило `INSTEAD`, так что предыдущее дерево запроса отбрасывается. Заметьте, что этот запрос по-прежнему использует представление `shoelace`. Но система правил ещё не закончила свою работу, она продолжает и применяет правило `_RETURN`, так что мы получаем:

```
UPDATE shoelace_data
   SET sl_name = s.sl_name,
       sl_avail = s.sl_avail + shoelace_arrive.arr_quant,
       sl_color = s.sl_color,
       sl_len = s.sl_len,
       sl_unit = s.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data,
       shoelace old, shoelace new,
       shoelace_data s, unit u
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name;
```

Наконец, применяется правило `log_shoelace` и выдаётся дополнительное дерево запроса:

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
```

```
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data,
       shoelace old, shoelace new,
       shoelace_data s, unit u,
       shoelace_data old, shoelace_data new
       shoelace_log shoelace_log
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND (s.sl_avail + shoelace_arrive.arr_quant) <> s.sl_avail;
```

Теперь, обработав все правила, система правил выдаёт построенные деревья запросов.

В итоге мы получаем два дерева запросов, равнозначные следующим операторам SQL:

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND s.sl_avail + shoelace_arrive.arr_quant <> s.sl_avail;

UPDATE shoelace_data
   SET sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive,
       shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.sl_name
   AND shoelace_data.sl_name = s.sl_name;
```

В результате вся операция, в ходе которой данные, поступающие из одного отношения, вставляются в другое, вставка преобразуется в изменение третьего, что затем становится изменением четвёртого, и запись об этом изменении добавляется в пятое, сводится к двум запросам.

Здесь можно заметить маленькую не очень красивую деталь. Как видно, в этих двух запросах таблица `shoelace_data` фигурирует в списке отношений дважды, тогда как определённо достаточно и одного вхождения. Планировщик не понимает этого и поэтому для дерева запроса `INSERT`, выданного системой правил, будет получен такой план:

```
Nested Loop
  ->  Merge Join
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on s
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on shoelace_arrive
  ->  Seq Scan on shoelace_data
```

Тогда как без лишнего элемента в списке отношений мы получили бы:

```
Merge Join
  ->  Seq Scan
        ->  Sort
```

```
              ->  Seq Scan on s
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on shoelace_arrive
```

При этом в журнале оказались бы точно такие же записи. Таким образом, применение правил повлекло дополнительное сканирование таблицы `shoelace_data`, в котором не было никакой необходимости. И такое же избыточное сканирование выполняется ещё раз в `UPDATE`. Отнеситесь к этому с пониманием, ведь сделать всё это возможным в принципе было действительно сложно.

И наконец, ещё одна, завершающая демонстрация системы правил Postgres Pro и всей её мощи. Предположим, что вы добавили в базу данных шнурки с экстраординарными цветами:

```
INSERT INTO shoelace VALUES ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
INSERT INTO shoelace VALUES ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

Давайте создадим представление, чтобы убедиться, что шнурки (записи в `shoelace`) не подходят ни к каким туфлям. Оно будет определено так:

```
CREATE VIEW shoelace_mismatch AS
    SELECT * FROM shoelace WHERE NOT EXISTS
        (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

Через него мы получаем наши записи:

```
SELECT * FROM shoelace_mismatch;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl9     |        0 | pink     |     35 | inch    |      88.9
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
```

Теперь мы хотим, чтобы шнурки, которые ни к чему не подходят, удалялись из базы данных. Чтобы немного усложнить задачу для Postgres Pro, мы не будем удалять их непосредственно из таблицы. Вместо этого мы создадим ещё одно представление:

```
CREATE VIEW shoelace_can_delete AS
    SELECT * FROM shoelace_mismatch WHERE sl_avail = 0;
```

И удалим их так:

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_can_delete
             WHERE sl_name = shoelace.sl_name);
```

Вуаля:

```
SELECT * FROM shoelace;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl1     |        5 | black    |     80 | cm      |        80
 sl2     |        6 | black    |    100 | cm      |       100
 sl7     |        6 | brown    |     60 | cm      |        60
 sl4     |        8 | black    |     40 | inch    |     101.6
 sl3     |       10 | black    |     35 | inch    |      88.9
 sl8     |       21 | brown    |     40 | inch    |     101.6
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
 sl5     |        4 | brown    |      1 | m       |       100
 sl6     |       20 | brown    |    0.9 | m       |        90
(9 rows)
```

Так запрос `DELETE` для представления с ограничивающим условием-подзапросом, использующим в совокупности 4 вложенных/соединённых представления, с одним из которых тоже связано

условие с подзапросом, задействующим представление, и где используются вычисляемые столбцы представлений, переписывается и преобразуется в одно дерево запроса, которое удаляет требуемые данные из реальной таблицы.

На практике ситуации, когда необходима такая сложная конструкция, встречаются довольно редко, но тем не менее приятно осознавать, что всё это возможно и работает.

38.5. Правила и права

В результате переписывания запросов системой правил Postgres Pro обращение может происходить не к тем таблицам/представлениям, к которым обращался исходный запрос. С правилами для изменения возможна так же и запись в другие таблицы.

Правила перезаписи не имеют отдельного владельца — владельцем правил перезаписи, определённых для отношения (таблицы или представления), автоматически считается владелец этого отношения. Система правил Postgres Pro меняет поведение стандартного механизма управления доступом. К отношениям, используемым вследствие применения правил, проверяется доступ владельца правила, но не пользователя, выполняющего запрос. Это значит, что пользователь должен иметь права, необходимые только для обращения к таблицам/представлениям, которые он явно упоминает в своих запросах.

Например, представим, что у пользователя есть список телефонных номеров, некоторые из которых личные, а некоторые должна знать его ассистентка. Он может построить следующую конструкцию:

```
CREATE TABLE phone_data (person text, phone text, private boolean);
CREATE VIEW phone_number AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data;
GRANT SELECT ON phone_number TO assistant;
```

Никто, кроме него (и суперпользователей базы данных) не сможет обратиться к таблице `phone_data`. Но так как ассистентке было дано (`GRANT`) соответствующее право, она сможет выполнить `SELECT` для представления `phone_number`. Система правил преобразует `SELECT` из `phone_number` в `SELECT` из таблицы `phone_data`. Так как пользователь является владельцем `phone_number`, он же считается владельцем правила, доступ на чтение `phone_data` проверяется для него, и выполнение запроса разрешается. Проверка прав доступа к `phone_number` тоже выполняется, но при этом проверяется пользователь, выполняющий запрос, так что обращаться к этому представлению смогут только сам пользователь и его ассистентка.

Права проверяются правило за правилом. То есть, в данный момент только ассистентка может видеть открытые телефонные номера. Но она может создать другое представление и дать доступ к нему всем (роли `public`), после чего все смогут видеть данные `phone_number` через представление ассистентки. Что она не может сделать, так это создать представление, которое обращается к `phone_data` напрямую. (Вообще она может это сделать, но такое представление не будет работать, так как при любой попытке прочитать его доступ к таблице будет запрещён.) И как только пользователь заметит, что ассистентка открыла доступ к своему представлению `phone_number`, он может лишить её права чтения этого представления. В результате все сразу потеряют доступ и к представлению ассистентки.

Может показаться, что такая проверка «правило-за-правилом» представляет уязвимость, но это не так. Если бы даже этот механизм не работал, ассистентка могла бы создать таблицу со столбцами как в `phone_number` и регулярно копировать туда данные. Тогда это были бы её собственные данные и она могла бы открывать доступ к ним кому угодно. Другими словами, команда `GRANT` означает «Я доверяю тебе». Если кто-то, кому вы доверяете, проделывает такие операции, стоит задуматься и, возможно, лишить его доступа к данным, применив `REVOKE`.

Хотя представления могут применяться для скрытия содержимого определённых столбцов, как описано выше, с их помощью нельзя надёжно скрыть данные в невидимых строках, если только не установлен флаг `security_barrier`. Например, следующее представление небезопасно:

```
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

Может показаться, что всё в порядке, ведь система правил преобразует SELECT из phone_number в SELECT из phone_data и добавит ограничивающее условие, чтобы выдавались только строки с полем phone, начинающимся не с 412. Но если пользователь может создавать собственные функции, ему будет не сложно заставить планировщик выполнять функцию пользователя перед выражением NOT LIKE. Например:

```
CREATE FUNCTION tricky(text, text) RETURNS bool AS $$
BEGIN
    RAISE NOTICE '% => %', $1, $2;
    RETURN true;
END;
$$ LANGUAGE plpgsql COST 0.0000000000000000000001;

SELECT * FROM phone_number WHERE tricky(person, phone);
```

Так он сможет получить все имена и номера телефонов из таблицы phone_data через сообщения NOTICE, так как планировщик решит, что лучше выполнить недорогую функцию tricky перед более дорогой операцией NOT LIKE. И даже если пользователь не имеет права создавать новые функции, он может использовать для подобных атак встроенные функции. (Например, многие функции приведения показывают входные значения в сообщениях об ошибках.)

Подобные соображения распространяются и на правила для изменения. Применительно к примерам предыдущего раздела, владелец таблиц в базе данных может дать кому-нибудь другому для представления shoelace права SELECT, INSERT, UPDATE и DELETE, а для shoelace_log только SELECT. Действие правила, добавляющее записи в журнал, всё равно будет выполняться успешно, а этот другой пользователь сможет видеть записи в журнале. Но он не сможет создавать поддельные записи, равно как и модифицировать или удалять существующие. В этом случае нет никакой возможности заставить планировщик изменить порядок операций, так как единственное правило, которое обращается к shoelace_log — это безусловный INSERT. В более сложных сценариях это может быть не так.

Когда требуется, чтобы представление обеспечивало защиту на уровне строк, к нему нужно применить атрибут security_barrier. Это предотвратит утечку содержимого строк из злонамеренно выбранных функций и операторов до того, как строки будут отфильтрованы представлением. Например, показанное выше представление будет безопасным, если создать его так:

```
CREATE VIEW phone_number WITH (security_barrier) AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

Представления, созданные с атрибутом security_barrier, могут работать гораздо медленнее, чем обычные. И вообще говоря, это неизбежно: самый быстрый план должен быть отвергнут, если он может скомпрометировать защиту. Поэтому данный атрибут по умолчанию не устанавливается.

Планировщик запросов имеет больше свободы, работая с функциями, лишёнными побочных эффектов. Такие функции называются герметичными (LEAKPROOF) и включают только простые часто используемые операторы, например, операторы равенства. Планировщик запросов может безопасно вычислять такие функции в любой момент выполнения запроса, так как при вызове их для строк, невидимых пользователю, не просочится никакая информация об этих строках. Более того, функции, которые не принимают аргументы или которым не передаются аргументы из представления с барьером безопасности, можно не помечать как LEAKPROOF, чтобы они вышли наружу, так как они никогда не получат данные из представления. И напротив, функции, которые могут вызвать ошибку в зависимости от значений аргументов (например, в случае переполнения или деления на ноль), герметичными не являются, и могут выдать существенную информацию о невидимых строках, если будут выполнены перед фильтрами строк.

Важно понимать, что даже представление, созданное с атрибутом security_barrier, остаётся безопасным только в том смысле, что содержимое невидимых строк не будет передаваться

потенциально небезопасным функциям. Но пользователь может собрать некоторые сведения о невидимых данных и другими способами; например, он может проанализировать план запроса, полученный с EXPLAIN, или замерить время выполнения запросов с этим представлением. Злоумышленник может сделать определённые выводы об объёме невидимых данных или даже получить некоторую информацию о распределении данных или наиболее частых значениях (так как всё это отражается в статистике для оптимизатора и, как следствие, влияет на время выполнения плана или даже на выбор плана). Если возможность атаки через скрытые каналы вызывает опасения, вероятно, будет разумным не предоставлять никакой доступ к этим данным.

38.6. Правила и статус команд

Сервер Postgres Pro возвращает строку состояния команды, например, INSERT 149592 1, для каждой получаемой команды. Это довольно прозрачно, когда не задействуются правила, но что произойдёт, если правила перезапишут запрос?

Правила влияют на состояния команды следующим образом:

- Если с запросом не связано безусловное правило INSTEAD, то выполняется заданный исходный запрос и его статус выдаётся как обычно. (Но если определены какие-то условные правила INSTEAD, к исходному запросу добавляется условие, обратное их условиям применения. Это может повлиять на число обрабатываемых строк и выводимый статус команды.)
- Если с запросом связано безусловное правило INSTEAD, исходный запрос не выполняется вовсе. В этом случае сервер возвратит статус команды от последнего запроса, вставленного правилом INSTEAD (условным или безусловным), и тип команды исходного запроса (INSERT, UPDATE или DELETE). Если правила не добавили подходящего запроса, в возвращённом статусе команды показывается исходный тип запроса и нули вместо количества строк и OID.

Программист может добиться, чтобы статус команды во втором случае устанавливало нужное правило INSTEAD, назначив ему имя, стоящее по алфавиту после других активных правил, чтобы это правило применялось последним.

38.7. Сравнение правил и триггеров

Многие вещи, которые можно сделать с помощью триггеров, можно также реализовать, используя систему правил Postgres Pro. Однако, используя правила, нельзя реализовать, например, некоторые типы ограничений, в частности, внешние ключи. Хотя можно определить правило с ограничивающим условием, которое будет преобразовать команду в NOTHING, если значение ключа не находится в другой таблице, но при этом неподходящие данные будут отбрасываться молча, а это не самый лучший вариант. Также, если требуется проверить правильность значений и, обнаружив неверное значение, выдать ошибку, это нужно делать в триггере.

В этой главе мы разберём использование правил для изменения представлений. Все правила, приведённые в примерах этой главы, можно также заменить триггерами INSTEAD OF для представлений. Написать такие триггеры часто бывает проще, чем разработать правила, особенно если для изменений применяется сложная логика.

Для тех задач, которые можно решить обоими способами, лучший выбирается в зависимости от характера использования базы данных. Следует учитывать, что триггер срабатывает для каждой обрабатываемой строки, а правило изменяет существующий запрос или создаёт ещё один. Поэтому, если один оператор обрабатывает сразу много строк, правило, добавляющее дополнительную команду, скорее всего, будет работать быстрее, чем триггер, который вызывается для каждой очередной строки и должен каждый раз определять, что с ней делать. Однако триггеры концептуально гораздо проще правил, и использовать их правильно новичкам гораздо проще.

Давайте рассмотрим пример, показывающий, как выбор в пользу правил вместо триггеров оказывается выигрышным в определённой ситуации. Пусть у нас есть две таблицы:

```
CREATE TABLE computer (
    hostname        text,    -- индексированное
```

```
    manufacturer    text     -- индексированное
);

CREATE TABLE software (
    software        text,    -- индексированное
    hostname        text     -- индексированное
);
```

Обе таблицы содержат несколько тысяч строк, а индексы по полю `hostname` являются уникальными. Правило или триггер должны реализовать ограничение, которое удалит строки из таблицы `software`, ссылающиеся на удаляемый компьютер. Триггер выполнял бы такую команду:

```
DELETE FROM software WHERE hostname = $1;
```

Так как триггер вызывается для каждой отдельной строки, удаляемой из таблицы `computer`, он может подготовить и сохранить план этой команды, а затем передавать значение `hostname` подготовленному запросу в параметрах. Правило же можно записать так:

```
CREATE RULE computer_del AS ON DELETE TO computer
    DO DELETE FROM software WHERE hostname = OLD.hostname;
```

Теперь давайте взглянем на разные варианты удаления. В этом случае:

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

таблица `computer` сканируется по индексу (быстро), и команда, выполняемая триггером, так же будет применять сканирование по индексу (тоже быстро). Дополнительной командой правила будет:

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
                       AND software.hostname = computer.hostname;
```

Так как созданы все необходимые индексы, планировщик создаст план

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

Таким образом, большого различия в скорости между реализациями с триггером и с правилом не будет.

Теперь мы хотим избавиться от 2000 компьютеров, у которых `hostname` начинается с `old`. Это можно сделать двумя командами. Первая:

```
DELETE FROM computer WHERE hostname >= 'old'
                       AND hostname <  'ole'
```

Правило преобразует её в:

```
DELETE FROM software WHERE computer.hostname >= 'old' AND computer.hostname < 'ole'
                       AND software.hostname = computer.hostname;
```

с планом:

```
Hash Join
  ->  Seq Scan on software
  ->  Hash
    ->  Index Scan using comp_hostidx on computer
```

С другой возможной командой:

```
DELETE FROM computer WHERE hostname ~ '^old';
```

для запроса, преобразованного правилом, получается следующий план:

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

Это показывает, что планировщик не понимает, что ограничение по hostname в computer можно также использовать для сканирования по индексу в software, когда несколько условий объединяются с помощью AND, что он успешно делает для варианта команды с регулярным выражением. Триггер будет вызываться для каждой из 2000 удаляемых записей о старых компьютерах, и это приведёт к одному сканированию индекса в таблице computer и 2000 сканированиям индекса в таблице software. Реализация с правилом делает это двумя командами, применяющими индексы. Будет ли правило быстрее при последовательном сканировании, зависит от общего размера таблицы software. С другой стороны, выполнение 2000 команд из триггера через менеджер SPI всё равно займёт время, даже если все блоки индекса вскоре окажутся в кеше.

В завершение взгляните на эту команду:

```
DELETE FROM computer WHERE manufacturer = 'bim';
```

Она также может привести к удалению множества строк из таблицы computer. Поэтому триггер снова пропустит через исполнитель такое же множество команд. Правило же выдаст следующую команду:

```
DELETE FROM software WHERE computer.manufacturer = 'bim'
                       AND software.hostname = computer.hostname;
```

План для этой команды снова будет содержать вложенный цикл по двум сканированиям индекса, но на этот раз с другим индексом таблицы computer:

```
Nestloop
  ->  Index Scan using comp_manufidx on computer
  ->  Index Scan using soft_hostidx on software
```

Во всех этих случаях дополнительные команды будут более-менее независимыми от числа затрагиваемых строк.

Таким образом, правила будут значительно медленнее триггеров, только если их действия приводят к образованию больших и плохо связанных соединений, когда планировщик оказывается бессилен.

Глава 39. Процедурные языки

Postgres Pro позволяет разрабатывать пользовательские функции не только на SQL и C, но и на других языках. Эти языки в целом называются *процедурными языками* (PL, Procedural Language). Если функция написана на процедурном языке, сервер баз данных сам по себе не знает, как интерпретировать её исходный текст. Вместо этого он передаёт эту задачу специальному обработчику, понимающему данный язык. Обработчик может либо выполнить всю работу по разбору, синтаксическому анализу, выполнению кода и т. д., либо действовать как «прослойка» между Postgres Pro и внешним исполнителем языка программирования. Сам обработчик представляет собой функцию на языке C, скомпилированную в виде разделяемого объекта и загружаемую по требованию, как и любая другая функция на C.

В настоящее время стандартный дистрибутив Postgres Pro включает четыре процедурных языка: PL/pgSQL (Глава 40), PL/Tcl (Глава 41), PL/Perl (Глава 42) и PL/Python (Глава 43). Существуют и другие процедурные языки, поддержка которых не включена в базовый дистрибутив. Информацию о них можно найти в Приложении H. Кроме того, пользователи могут реализовать и другие языки; основы разработки нового процедурного языка рассматриваются в Главе 51.

39.1. Установка процедурных языков

Прежде всего, процедурный язык должен быть «установлен» в каждую базу данных, где он будет использоваться. Но процедурные языки, устанавливаемые в базу данных `template1`, автоматически становятся доступными во всех впоследствии создаваемых базах, так как их определения в `template1` будут скопированы командой `CREATE DATABASE`. Таким образом, администратор баз данных может выбрать, какие языки будут доступны в определённых базах данных, и при желании сделать некоторые языки доступными по умолчанию.

Для языков, включённых в стандартный дистрибутив, достаточно выполнить команду `CREATE EXTENSION` *`имя_языка`*, чтобы установить язык в текущую базу данных. Описанная ниже ручная процедура рекомендуется только для установки языков, не упакованных в виде расширений.

Установка процедурного языка вручную

Процедурный язык устанавливается в базу данных в пять этапов, и выполнять их должен администратор баз данных. В большинстве случаев необходимые команды SQL следует упаковать в виде установочного скрипта «расширения», чтобы их можно было выполнить, воспользовавшись командой `CREATE EXTENSION`.

1. Разделяемый объект для обработчика языка должен быть скомпилирован и установлен в соответствующий каталог библиотек. Это в принципе не отличается от сборки и установки дополнительных модулей с обычными функциями на языке C; см. Подраздел 35.9.5. Часто обработчик языка зависит от внешней библиотеки, в которой собственно реализован исполнитель языка программирования; в таких случаях нужно установить и эту библиотеку.

2. Обработчик должен быть объявлен командой

   ```
   CREATE FUNCTION имя_функции_обработчика()
       RETURNS language_handler
       AS 'путь-к-разделяемому-объекту'
       LANGUAGE C;
   ```

   Специальный тип возврата `language_handler` говорит СУБД, что эта функция не возвращает какой-либо определённый тип данных SQL, и значит её нельзя использовать непосредственно в операторах SQL.

3. (Optional) Дополнительно обработчик языка может предоставить функцию обработки «внедрённого кода», которая будет выполнять анонимные блоки кода (команды DO), написанные на этом языке. Если для языка есть обработчик внедрённого кода, объявите его такой командой:

   ```
   CREATE FUNCTION имя_обработчика_внедрённого_кода(internal)
   ```

```
    RETURNS void
    AS 'путь-к-разделяемому-объекту'
    LANGUAGE C;
```

4. (Optional) Кроме того, обработчик языка может предоставить функцию «проверки», которая будет проверять корректность определения функции, собственно не выполняя её. Функция проверки, если она существует, вызывается командой `CREATE FUNCTION`. Если такая функция для языка определена, объявите её такой командой:

   ```
   CREATE FUNCTION имя_функции_проверки(oid)
       RETURNS void
       AS 'путь-к-разделяемому-объекту'
       LANGUAGE C STRICT;
   ```

5. Наконец, процедурный язык должен быть объявлен командой

   ```
   CREATE [TRUSTED] [PROCEDURAL] LANGUAGE имя_языка
       HANDLER имя_функции_обработчика
       [INLINE имя_обработчика_внедрённого_кода]
       [VALIDATOR имя_функции_проверки] ;
   ```

   Необязательное ключевое слово `TRUSTED` (доверенный) указывает, что язык не предоставляет пользователю доступ к данным, которого он не имел бы без него. Доверенные языки предназначены для обычных пользователей баз данных, не имеющих прав суперпользователя, и их можно использовать для безопасного создания функций и триггерных процедур. Так как функции PL выполняются внутри сервера баз данных, флаг `TRUSTED` следует устанавливать только для тех языков, которые не позволяют обращаться к внутренним механизмам сервера или файловой системе. Языки PL/pgSQL, PL/Tcl и PL/Perl считаются доверенными; языки PL/TclU, PL/PerlU и PL/PythonU предоставляют неограниченную функциональность, и их *не* следует помечать как доверенные.

Примере 39.1 показывает, как выполняется процедура ручной установки для языка PL/Perl.

Пример 39.1. Установка PL/Perl вручную

Следующая команда говорит серверу баз данных, где найти разделяемый объект для функции-обработчика языка PL/Perl:

```
CREATE FUNCTION plperl_call_handler() RETURNS language_handler AS
    '$libdir/plperl' LANGUAGE C;
```

Для PL/Perl реализованы обработчик внедрённого кода и функция проверки, так что их мы тоже объявим:

```
CREATE FUNCTION plperl_inline_handler(internal) RETURNS void AS
    '$libdir/plperl' LANGUAGE C;

CREATE FUNCTION plperl_validator(oid) RETURNS void AS
    '$libdir/plperl' LANGUAGE C STRICT;
```

Следующая команда:

```
CREATE TRUSTED PROCEDURAL LANGUAGE plperl
    HANDLER plperl_call_handler
    INLINE plperl_inline_handler
    VALIDATOR plperl_validator;
```

определяет, что ранее объявленные функции должны вызываться для функций и триггерных процедур с атрибутом языка `plperl`.

В стандартной инсталляции Postgres Pro обработчик языка PL/pgSQL уже собран и установлен в каталог «библиотек»; более того, сам язык PL/pgSQL установлен во всех базах данных. Если при сборке сконфигурирована поддержка Tcl, то обработчики для PL/Tcl и PL/TclU собираются и устанавливаются в каталог библиотек, но сам язык по умолчанию в базы данных

не устанавливается. Подобным образом, если сконфигурирована поддержка Perl, собираются и устанавливаются обработчики PL/Perl и PL/PerlU, а при включении поддержки Python устанавливается обработчик PL/PythonU, но в базы данных эти языки по умолчанию не устанавливаются.

Глава 40. PL/pgSQL — процедурный язык SQL

40.1. Обзор

PL/pgSQL это процедурный язык для СУБД Postgres Pro. Целью проектирования PL/pgSQL было создание загружаемого процедурного языка, который:

- используется для создания функций и триггеров,
- добавляет управляющие структуры к языку SQL,
- может выполнять сложные вычисления,
- наследует все пользовательские типы, функции и операторы,
- может быть определён как доверенный язык,
- прост в использовании.

Функции PL/pgSQL могут использоваться везде, где допустимы встроенные функции. Например, можно создать функции со сложными вычислениями и условной логикой, а затем использовать их при определении операторов или в индексных выражениях.

В версии PostgreSQL 9.0 и выше PL/pgSQL устанавливается по умолчанию. Тем не менее это по-прежнему загружаемый модуль и администраторы, особо заботящиеся о безопасности, могут удалить его при необходимости.

40.1.1. Преимущества использования PL/pgSQL

Postgres Pro и большинство других СУБД используют SQL в качестве языка запросов. SQL хорошо переносим и прост в изучении. Однако каждый оператор SQL выполняется индивидуально на сервере базы данных.

Это значит, что ваше клиентское приложение должно каждый запрос отправлять на сервер, ждать пока он будет обработан, получать результат, делать некоторые вычисления, затем отправлять последующие запросы на сервер. Всё это требует межпроцессного взаимодействия, а также несёт нагрузку на сеть, если клиент и сервер базы данных расположены на разных компьютерах.

PL/pgSQL позволяет сгруппировать блок вычислений и последовательность запросов *внутри* сервера базы данных, таким образом, мы получаем силу процедурного языка и простоту использования SQL при значительной экономии накладных расходов на клиент-серверное взаимодействие.

- Исключаются дополнительные обращения между клиентом и сервером
- Промежуточные ненужные результаты не передаются между сервером и клиентом
- Есть возможность избежать многочисленных разборов одного запроса

В результате это приводит к значительному увеличению производительности по сравнению с приложением, которое не использует хранимых функций.

Кроме того, PL/pgSQL позволяет использовать все типы данных, операторы и функции SQL.

40.1.2. Поддерживаемые типы данных аргументов и возвращаемых значений

Функции на PL/pgSQL могут принимать в качестве аргументов все поддерживаемые сервером скалярные типы данных или массивы и возвращать в качестве результата любой из этих типов. Они могут принимать и возвращать любой именованный составной тип (тип кортежа). Также есть возможность объявить функцию на PL/pgSQL как возвращающую `record`, то есть её результатом

будет тип кортежа, столбцы которого определит спецификация вызывающего запроса, как описано в Подразделе 7.2.1.4.

Использование маркера VARIADIC позволяет объявлять функции на PL/pgSQL с переменным числом аргументов. Это работает точно так же, как и для функций на SQL, как описано в Подразделе 35.4.5.

Функции на PL/pgSQL могут принимать и возвращать полиморфные типы anyelement, anyarray, anynonarray, anyenum и anyrange. В таких случаях фактические типы данных могут меняться от вызова к вызову, как описано в Подраздел 35.2.5. Пример показан в Подразделе 40.3.1.

Функции на PL/pgSQL могут возвращать «множества» (или таблицы) любого типа, которые могут быть возвращены в виде одного объекта. Такие функции генерируют вывод, выполняя команду RETURN NEXT для каждого элемента результирующего набора или RETURN QUERY для вывода результата запроса.

Наконец, при отсутствии полезного возвращаемого значения функция на PL/pgSQL может возвращать void.

Функции на PL/pgSQL можно объявить с выходными параметрами вместо явного задания типа возвращаемого значения. Это не добавляет никаких фундаментальных возможностей языку, но часто бывает удобно, особенно для возвращения нескольких значений. Нотация RETURNS TABLE может использоваться вместо RETURNS SETOF.

Конкретные примеры рассматриваются в Подразделе 40.3.1 и Подразделе 40.6.1.

40.2. Структура PL/pgSQL

Функции, написанные на PL/pgSQL, определяются на сервере командами CREATE FUNCTION. Такая команда обычно выглядит, например, так:

```
CREATE FUNCTION somefunc(integer, text) RETURNS integer
AS 'тело функции'
LANGUAGE plpgsql;
```

Если рассматривать CREATE FUNCTION, тело функции представляет собой просто текстовую строку. Часто для написания тела функции удобнее заключать эту строку в доллары (см. Подраздел 4.1.2.4), а не в обычные апострофы. Если не применять заключение в доллары, все апострофы или обратные косые черты в теле функции придётся экранировать, дублируя их. Почти во всех примерах в этой главе тело функций заключается в доллары.

PL/pgSQL это блочно-структурированный язык. Текст тела функции должен быть *блоком*. Структура блока:

```
[ <<метка>> ]
[ DECLARE
    объявления ]
BEGIN
    операторы
END [ метка ];
```

Каждое объявление и каждый оператор в блоке должны завершаться символом «;» (точка с запятой). Блок, вложенный в другой блок, должен иметь точку с запятой после END, как показано выше. Однако финальный END, завершающий тело функции, не требует точки с запятой.

> **Подсказка**
>
> Распространённой ошибкой является добавление точки с запятой сразу после BEGIN. Это неправильно и приведёт к синтаксической ошибке.

Метка требуется только тогда, когда нужно идентифицировать блок в операторе EXIT, или дополнить имена переменных, объявленных в этом блоке. Если метка указана после END, то она должна совпадать с меткой в начале блока.

Ключевые слова не чувствительны к регистру символов. Как и в обычных SQL-командах, идентификаторы неявно преобразуются к нижнему регистру, если они не взяты в двойные кавычки.

Комментарии в PL/pgSQL коде работают так же, как и в обычном SQL. Двойное тире (--) начинает комментарий, который завершается в конце строки. Блочный комментарий начинается с /* и завершается */. Блочные комментарии могут быть вложенными.

Любой оператор в выполняемой секции блока может быть *вложенным блоком*. Вложенные блоки используются для логической группировки нескольких операторов или локализации области действия переменных для группы операторов. Во время выполнения вложенного блока переменные, объявленные в нём, скрывают переменные внешних блоков с такими же именами. Чтобы получить доступ к внешним переменным, нужно дополнить их имена меткой блока. Например:

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Сейчас quantity = %', quantity;  -- Выводится 30
    quantity := 50;
    --
    -- Вложенный блок
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Сейчас quantity = %', quantity;  -- Выводится 80
        RAISE NOTICE 'Во внешнем блоке quantity = %', outerblock.quantity;  --
 Выводится 50
    END;

    RAISE NOTICE 'Сейчас quantity = %', quantity;  -- Выводится 50

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

> **Примечание**
>
> Существует скрытый «внешний блок», окружающий тело каждой функции на PL/pgSQL. Этот блок содержит объявления параметров функции (если они есть), а также некоторые специальные переменные, такие как FOUND (см. Подраздел 40.5.5). Этот блок имеет метку, совпадающую с именем функции, таким образом, параметры и специальные переменные могут быть дополнены именем функции.

Важно не путать использование BEGIN/END для группировки операторов в PL/pgSQL с одноимёнными SQL-командами для управления транзакциями. BEGIN/END в PL/pgSQL служат только для группировки предложений; они не начинают и не заканчивают транзакции. Функции и триггерные процедуры всегда выполняются в рамках транзакции, начатой во внешнем запросе — они не могут начать или завершить эту транзакцию, так как у них внутри нет контекста для выполнения таких действий. Однако блок содержащий секцию EXCEPTION создаёт вложенную

транзакцию, которая может быть отменена, не затрагивая внешней транзакции. Подробнее это описано в Подразделе 40.6.6.

40.3. Объявления

Все переменные, используемые в блоке, должны быть определены в секции объявления. (За исключением переменной-счётчика цикла FOR, которая объявляется автоматически. Для цикла по диапазону чисел автоматически объявляется целочисленная переменная, а для цикла по результатам курсора - переменная типа record.)

Переменные PL/pgSQL могут иметь любой тип данных SQL, такой как integer, varchar, char.

Примеры объявления переменных:

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

Общий синтаксис объявления переменной:

```
имя [ CONSTANT ] тип [ COLLATE имя_правила_сортировки ] [ NOT NULL ] [ { DEFAULT | := |
 = } выражение ];
```

Предложение DEFAULT, если присутствует, задаёт начальное значение, которое присваивается переменной при входе в блок. Если отсутствует, то переменная инициализируется SQL-значением NULL. Указание CONSTANT предотвращает изменение значения переменной после инициализации, таким образом, значение остаётся постоянным в течение всего блока. Параметр COLLATE определяет правило сортировки, которое будет использоваться для этой переменной (см. Подраздел 40.3.6). Если указано NOT NULL, то попытка присвоить NULL во время выполнения приведёт к ошибке. Все переменные, объявленные как NOT NULL, должны иметь непустые значения по умолчанию. Можно использовать знак равенства (=) вместо совместимого с PL/SQL :=.

Значение по умолчанию вычисляется и присваивается переменной каждый раз при входе в блок (не только при первом вызове функции). Так, например, если переменная типа timestamp имеет функцию now() в качестве значения по умолчанию, это приведёт к тому, что переменная всегда будет содержать время текущего вызова функции, а не время, когда функция была предварительно скомпилирована.

Примеры:

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

40.3.1. Объявление параметров функции

Переданные в функцию параметры именуются идентификаторами $1, $2 и т. д. Дополнительно, для улучшения читаемости, можно объявить псевдонимы для параметров $*n*. Либо псевдоним, либо цифровой идентификатор используются для обозначения параметра.

Создать псевдоним можно двумя способами. Предпочтительный способ это дать имя параметру в команде CREATE FUNCTION, например:

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
```

```
$$ LANGUAGE plpgsql;
```

Другой способ это явное объявление псевдонима при помощи синтаксиса:

```
имя ALIAS FOR $n;
```

Предыдущий пример для этого стиля выглядит так:

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

> **Примечание**
>
> Эти два примера не полностью эквивалентны. В первом случае на `subtotal` можно ссылаться как `sales_tax.subtotal`, а во втором случае такая ссылка невозможна. (Если бы к внутреннему блоку была добавлена метка, то `subtotal` можно было бы дополнить этой меткой.)

Ещё несколько примеров:

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- вычисления, использующие v_string и index
END;
$$ LANGUAGE plpgsql;


CREATE FUNCTION concat_selected_fields(in_t sometablename) RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

Когда функция на PL/pgSQL объявляется с выходными параметрами, им выдаются цифровые идентификаторы `$n` и для них можно создавать псевдонимы точно таким же способом, как и для обычных входных параметров. Выходной параметр это фактически переменная, стартующая с NULL и которой присваивается значение во время выполнения функции. Возвращается последнее присвоенное значение. Например, функция sales_tax может быть переписана так:

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Обратите внимание, что мы опустили `RETURNS real` — хотя можно было и включить, но это было бы излишним.

Выходные параметры наиболее полезны для возвращения нескольких значений. Простейший пример:

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
```

```
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

Как обсуждалось в Подразделе 35.4.4, здесь фактически создаётся анонимный тип `record` для возвращения результата функции. Если используется предложение `RETURNS`, то оно должно выглядеть как `RETURNS record`.

Есть ещё способ объявить функцию на PL/pgSQL с использованием `RETURNS TABLE`, например:

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT s.quantity, s.quantity * s.price FROM sales s
                 WHERE s.itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

Это в точности соответствует объявлению одного или нескольких параметров `OUT` и указанию `RETURNS SETOF` *некий_тип*.

Для функции на PL/pgSQL, возвращающей полиморфный тип (`anyelement`, `anyarray`, `anynonarray`, `anyenum`, `anyrange`), создаётся специальный параметр `$0`. Его тип данных соответствует типу, фактически возвращаемому функцией, и который устанавливается на основании фактических типов входных параметров (см. Подраздел 35.2.5). Это позволяет функции обращаться к фактически возвращаемому типу данных, как показано в Подразделе 40.3.3. Параметр `$0` инициализируется в `NULL` и его можно изменять внутри функции. Таким образом, его можно использовать для хранения возвращаемого значения, хотя это необязательно. Параметру `$0` можно дать псевдоним. В следующем примере функция работает с любым типом данных, поддерживающим оператор `+`:

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

Такой же эффект получается при объявлении одного или нескольких выходных параметров полиморфного типа. При этом `$0` не создаётся; выходные параметры сами используются для этой цели. Например:

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement,
                                 OUT sum anyelement)
AS $$
BEGIN
    sum := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

40.3.2. ALIAS

новое_имя `ALIAS FOR` *старое_имя*;

Синтаксис `ALIAS` более общий, чем предполагалось в предыдущем разделе: псевдонимы можно объявлять для любых переменных, а не только для параметров функции. Основная практическая

польза в том, чтобы назначить другие имена переменным с предопределёнными названиями, таким как NEW или OLD в триггерной процедуре.

Примеры:

```
DECLARE
  prior ALIAS FOR old;
  updated ALIAS FOR new;
```

Поскольку ALIAS даёт два различных способа именования одних и тех же объектов, то его неограниченное использование может привести к путанице. Лучше всего использовать ALIAS для переименования предопределённых имён.

40.3.3. Наследование типов данных

переменная%TYPE

Конструкция %TYPE предоставляет тип данных переменной или столбца таблицы. Её можно использовать для объявления переменных, содержащих значения из базы данных. Например, для объявления переменной с таким же типом, как и столбец user_id в таблице users нужно написать:

```
user_id users.user_id%TYPE;
```

Используя %TYPE, не нужно знать тип данных структуры, на которую вы ссылаетесь. И самое главное, если в будущем тип данных изменится (например: тип данных для user_id поменяется с integer на real), то вам может не понадобиться изменять определение функции.

Использование %TYPE особенно полезно в полиморфных функциях, поскольку типы данных, необходимые для внутренних переменных, могут меняться от одного вызова к другому. Соответствующие переменные могут быть созданы с применением %TYPE к аргументам и возвращаемому значению функции.

40.3.4. Типы кортежей

имя имя_таблицы%ROWTYPE;
имя имя_составного_типа;

Переменная составного типа называется *переменной-кортежем* (или переменной *типа кортежа*). Значением такой переменной может быть целый кортеж, полученный в результате выполнения запроса SELECT или FOR, при условии, что набор столбцов запроса соответствует заявленному типу переменной. Доступ к отдельным полям значения кортежа осуществляется как обычно, через точку, например rowvar.field.

Переменная-кортеж может быть объявлена с таким же типом, как и строка в существующей таблице или представлении, используя нотацию *имя_таблицы*%ROWTYPE; или с именем составного типа. (Поскольку каждая таблица имеет соответствующий составной тип с таким же именем, то на самом деле в Postgres Pro не имеет значения, пишете ли вы %ROWTYPE или нет. Но использование %ROWTYPE более переносимо.)

Параметры функции могут быть составного типа (строки таблицы). В этом случае соответствующий идентификатор $*n* будет переменной-кортежем, поля которой можно выбирать, например $1.user_id.

Только определённые пользователем столбцы таблицы доступны в переменной типа кортежа, но не OID или другие системные столбцы (потому что это может быть строка представления). Поля в таком типе наследуют размер и точность от типов данных столбцов таблицы, таких как char(*n*).

Ниже приведён пример использования составных типов. table1 и table2 это существующие таблицы, имеющие, по меньшей мере, перечисленные столбцы:

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
```

```
    t2_row table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2 WHERE ... ;
    RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```

40.3.5. Тип `record`

имя RECORD;

Переменные типа `record` похожи на переменные-кортежи, но они не имеют предопределённой структуры. Они приобретают фактическую структуру от строки, которая им присваивается командами `SELECT` или `FOR`. Структура переменной типа `record` может меняться каждый раз при присваивании значения. Следствием этого является то, что пока значение не присвоено первый раз, переменная типа `record` не имеет структуры и любая попытка получить доступ к отдельному полю приведёт к ошибке во время выполнения.

Обратите внимание, что `RECORD` это не подлинный тип данных, а только лишь заполнитель. Также следует понимать, что функция на PL/pgSQL, имеющая тип возвращаемого значения `record`, это не то же самое, что и переменная типа `record`, хотя такая функция может использовать переменную типа `record` для хранения своего результата. В обоих случаях фактическая структура строки неизвестна во время создания функции, но для функции, возвращающей `record`, фактическая структура определяется во время разбора вызывающего запроса, в то время как переменная типа `record` может менять свою структуру на лету.

40.3.6. Упорядочение переменных PL/pgSQL

Когда функция на PL/pgSQL имеет один или несколько параметров сортируемых типов данных, правило сортировки определяется при каждом вызове функции в зависимости от правил сортировки фактических аргументов, как описано в Разделе 22.2. Если оно определено успешно (т. е. среди аргументов нет конфликтов между неявными правилами сортировки), то все соответствующие параметры неявно трактуются как имеющее это правило сортировки. Внутри функции это будет влиять на поведение операторов, зависящих от используемого правила сортировки. Рассмотрим пример:

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b;
END;
$$ LANGUAGE plpgsql;

SELECT less_than(text_field_1, text_field_2) FROM table1;
SELECT less_than(text_field_1, text_field_2 COLLATE "C") FROM table1;
```

В первом случае `less_than` будет использовать для сравнения общее правило сортировки для `text_field_1` и `text_field_2`, в то время как во втором случае будет использоваться правило `C`.

Кроме того, определённое для вызова функции правило сортировки также будет использоваться для любых локальных переменных соответствующего типа. Таким образом, функция не станет работать по-другому, если её переписать так:

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
DECLARE
    local_a text := a;
    local_b text := b;
BEGIN
    RETURN local_a < local_b;
END;
```

```
$$ LANGUAGE plpgsql;
```

Если параметров с типами данных, поддерживающими сортировку, нет, или для параметров невозможно определить общее правило сортировки, тогда для параметров и локальных переменных применяются правила, принятые для их типа данных по умолчанию (которые обычно совпадают с правилами сортировки по умолчанию, принятыми для базы данных, но могут отличаться для переменных доменных типов).

Локальная переменная может иметь правило сортировки, отличное от правила по умолчанию. Для этого используется параметр `COLLATE` в объявлении переменной, например:

```
DECLARE
    local_a text COLLATE "en_US";
```

Этот параметр переопределяет правило сортировки, которое получила бы переменная в соответствии с вышеуказанными правилами.

И, конечно же, можно явно указывать параметр `COLLATE` для конкретных операций внутри функции, если к ним требуется применить конкретное правило сортировки. Например:

```
CREATE FUNCTION less_than_c(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b COLLATE "C";
END;
$$ LANGUAGE plpgsql;
```

Как и в обычной SQL-команде, это переопределяет правила сортировки, связанные с полями таблицы, параметрами и локальными переменными, которые используются в данном выражении.

40.4. Выражения

Все выражения, используемые в операторах PL/pgSQL, обрабатываются основным исполнителем SQL-сервера. Например, для вычисления такого выражения:

```
IF выражение THEN ...
```

PL/pgSQL отправит следующий запрос исполнителю SQL:

```
SELECT выражение
```

При формировании команды `SELECT` все вхождения имён переменных PL/pgSQL заменяются параметрами, как подробно описано в Подразделе 40.10.1. Это позволяет один раз подготовить план выполнения команды `SELECT` и повторно использовать его в последующих вычислениях с различными значениями переменных. Таким образом, при первом использовании выражения, по сути происходит выполнение команды `PREPARE`. Например, если мы объявили две целочисленные переменные `x` и `y`, и написали:

```
IF x < y THEN ...
```

то, что реально происходит за сценой, эквивалентно:

```
PREPARE имя_оператора(integer, integer) AS SELECT $1 < $2;
```

и затем, эта подготовленная команда исполняется (`EXECUTE`) для каждого оператора `IF` с текущими значениями переменных PL/pgSQL, переданных как значения параметров. Обычно эти детали не важны для пользователей PL/pgSQL, но их полезно знать при диагностировании проблем. Более подробно об этом рассказывается в Подразделе 40.10.2.

40.5. Основные операторы

В этом и последующих разделах описаны все типы операторов, которые понимает PL/pgSQL. Все, что не признается в качестве одного из этих типов операторов, считается командой SQL и отправляется для исполнения в основную машину базы данных, как описано в Подразделе 40.5.2 и Подразделе 40.5.3.

40.5.1. Присваивания

Присваивание значения переменной PL/pgSQL записывается в виде:

```
переменная { := | = } выражение;
```

Как описывалось ранее, выражение в таком операторе вычисляется с помощью SQL-команды SELECT, посылаемой в основную машину базы данных. Выражение должно получить одно значение (возможно, значение строки, если это переменная-кортеж или переменная типа record). Целевая переменная может быть простой переменной (возможно, дополненной именем блока), полем кортежа или записи; или элементом массива, который является простой переменной или полем. Для присваивания можно использовать знак равенства (=) вместо совместимого с PL/SQL :=.

Если тип данных результата выражения не соответствует типу данных переменной, это значение будет преобразовано к нужному типу с использованием приведения присваивания (см. Раздел 10.4). В случае отсутствия приведения присваивания для этой пары типов, интерпретатор PL/pgSQL попытается преобразовать значение результата через текстовый формат, то есть применив функцию вывода типа результата, а за ней функцию ввода типа переменной. Заметьте, что при этом функция ввода может выдавать ошибки времени выполнения, если не воспримет строковое представление значения результата.

Примеры:

```
tax := subtotal * 0.06;
my_record.user_id := 20;
```

40.5.2. Выполнение команды, не возвращающей результат

В функции на PL/pgSQL можно выполнить любую команду SQL, не возвращающую строк, просто написав эту команду (например, INSERT без предложения RETURNING).

Имя любой переменной PL/pgSQL в тексте команды рассматривается как параметр, а затем текущее значение переменной подставляется в качестве значения параметра во время выполнения. Это в точности совпадает с описанной ранее обработкой для выражений; за подробностями обратитесь к Подразделу 40.10.1.

При выполнении SQL-команды таким образом, PL/pgSQL может кешировать и повторно использовать план выполнения команды, как обсуждается в Подразделе 40.10.2.

Иногда бывает полезно вычислить значение выражения или запроса SELECT, но отказаться от результата, например, при вызове функции, у которой есть побочные эффекты, но нет полезного результата. Для этого в PL/pgSQL, используется оператор PERFORM:

```
PERFORM запрос;
```

Эта команда выполняет *запрос* и отбрасывает результат. *Запросы* пишутся таким же образом, как и в команде SQL SELECT, но ключевое слово SELECT заменяется на PERFORM. Для запросов WITH после PERFORM нужно поместить запрос в скобки. (В этом случае запрос может вернуть только одну строку.) Переменные PL/pgSQL будут подставлены в запрос так же, как и в команду, не возвращающую результат, план запроса также кешируется. Кроме того, специальная переменная FOUND устанавливается в истину, если запрос возвращает, по крайней мере, одну строку, или ложь, если не возвращает ни одной строки (см. Подраздел 40.5.5).

> **Примечание**
>
> Можно предположить, что такой же результат получается непосредственно командой SELECT, но в настоящее время использование PERFORM является единственным способом. Команда SQL, которая может возвращать строки, например SELECT, будет отклонена с ошибкой, если не имеет предложения INTO, как описано в следующем разделе.

Пример:

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

40.5.3. Выполнение запроса, возвращающего одну строку

Результат SQL-команды, возвращающей одну строку (возможно из нескольких столбцов), может быть присвоен переменной типа `record`, переменной-кортежу или списку скалярных переменных. Для этого нужно к основной команде SQL добавить предложение `INTO`. Так, например:

```
SELECT выражения_select INTO [STRICT] цель FROM ...;
INSERT ... RETURNING выражения INTO [STRICT] цель;
UPDATE ... RETURNING выражения INTO [STRICT] цель;
DELETE ... RETURNING выражения INTO [STRICT] цель;
```

где *цель* может быть переменной типа `record`, переменной-кортежем или разделённым запятыми списком скалярных переменных, полей записи/строки. Переменные PL/pgSQL подставляются в оставшуюся часть запроса, план выполнения кешируется, так же, как было описано выше для команд, не возвращающих строки. Это работает для команд `SELECT`, `INSERT`/`UPDATE`/`DELETE` с предложением `RETURNING` и служебных команд, возвращающих результат в виде набора строк (таких как `EXPLAIN`). За исключением предложения `INTO`, это те же SQL-команды, как их можно написать вне PL/pgSQL.

> **Подсказка**
>
> Обратите внимание, что данная интерпретация `SELECT` с `INTO` полностью отличается от Postgres Pro команды `SELECT INTO`, где в `INTO` указывается вновь создаваемая таблица. Если вы хотите в функции на PL/pgSQL создать таблицу, основанную на результате команды `SELECT`, используйте синтаксис `CREATE TABLE ... AS SELECT`.

Если результат запроса присваивается кортежу или списку переменных, то они должны в точности соответствовать по количеству и типам данных столбцам результата, иначе произойдёт ошибка во время выполнения. Если используется переменная типа `record`, то она автоматически приводится к типу строки результата запроса.

Предложение INTO может появиться практически в любом месте SQL-команды. Обычно его записывают непосредственно перед или сразу после списка *выражения_select* в `SELECT` или в конце команды для команд других типов. Рекомендуется следовать этому соглашению на случай, если правила разбора PL/pgSQL ужесточатся в будущих версиях.

Если указание `STRICT` отсутствует в предложении `INTO`, то *цели* присваивается первая строка, возвращённая запросом; или NULL, если запрос не вернул строк. (Заметим, что понятие «первая строка» определяется неоднозначно без `ORDER BY`.) Все остальные строки результата после первой отбрасываются. Можно проверить специальную переменную `FOUND` (см. Подраздел 40.5.5), чтобы определить, была ли возвращена запись:

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION 'Сотрудник % не найден', myname;
END IF;
```

Если добавлено указание `STRICT`, то запрос должен вернуть ровно одну строку или произойдёт ошибка во время выполнения: либо `NO_DATA_FOUND` (нет строк), либо `TOO_MANY_ROWS` (более одной строки). Можно использовать секцию исключений в блоке для обработки ошибок, например:

```
BEGIN
    SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
```

```
            RAISE EXCEPTION 'Сотрудник % не найден', myname;
        WHEN TOO_MANY_ROWS THEN
            RAISE EXCEPTION 'Сотрудник % уже существует', myname;
END;
```

После успешного выполнения команды с указанием STRICT, значение переменной FOUND всегда устанавливается в истину.

Для INSERT/UPDATE/DELETE с RETURNING, PL/pgSQL возвращает ошибку, если выбрано более одной строки, даже в том случае, когда указание STRICT отсутствует. Так происходит потому, что у этих команд нет возможности, типа ORDER BY, указать какая из задействованных строк должна быть возвращена.

Если для функции включён режим print_strict_params, то при возникновении ошибки, связанной с нарушением условия STRICT, в детальную (DETAIL) часть сообщения об ошибке будет включена информация о параметрах, переданных запросу. Изменить значение print_strict_params можно установкой параметра plpgsql.print_strict_params. Но это повлияет только на функции, скомпилированные после изменения. Для конкретной функции можно использовать указание компилятора, например:

```
CREATE FUNCTION get_userid(username text) RETURNS int
AS $$
#print_strict_params on
DECLARE
userid int;
BEGIN
    SELECT users.userid INTO STRICT userid
        FROM users WHERE users.username = get_userid.username;
    RETURN userid;
END;
$$ LANGUAGE plpgsql;
```

В случае сбоя будет сформировано примерно такое сообщение об ошибке

```
ERROR:  query returned no rows
DETAIL:  parameters: $1 = 'nosuchuser'
CONTEXT:  PL/pgSQL function get_userid(text) line 6 at SQL statement
```

> **Примечание**
>
> С указанием STRICT поведение SELECT INTO и связанных операторов соответствует принятому в Oracle PL/SQL.

Как действовать в случаях, когда требуется обработать несколько строк результата, описано в Подразделе 40.6.4.

40.5.4. Выполнение динамически формируемых команд

Часто требуется динамически формировать команды внутри функций на PL/pgSQL, то есть такие команды, в которых при каждом выполнении могут использоваться разные таблицы или типы данных. Обычно PL/pgSQL кеширует планы выполнения (как описано в Подразделе 40.10.2), но в случае с динамическими командами это не будет работать. Для исполнения динамических команд предусмотрен оператор EXECUTE:

```
EXECUTE строка-команды [ INTO [STRICT] цель ] [ USING выражение [, ... ] ];
```

где *строка-команды* это выражение, формирующее строку (типа text) с текстом команды, которую нужно выполнить. Необязательная *цель* — это переменная-запись, переменная-кортеж или разделённый запятыми список простых переменных и полей записи/кортежа, куда будут

помещены результаты команды. Необязательные выражения в USING формируют значения, которые будут вставлены в команду.

В сформированном тексте команды замена имён переменных PL/pgSQL на их значения проводиться не будет. Все необходимые значения переменных должны быть вставлены в командную строку при её построении, либо нужно использовать параметры, как описано ниже.

Также, нет никакого плана кеширования для команд, выполняемых с помощью EXECUTE. Вместо этого план создаётся каждый раз при выполнении. Таким образом, строка команды может динамически создаваться внутри функции для выполнения действий с различными таблицами и столбцами.

Предложение INTO указывает, куда должны быть помещены результаты SQL-команды, возвращающей строки. Если передаётся кортеж или список переменных, то они должны в точности соответствовать структуре результата запроса (когда используется переменная типа record, она автоматически приводится к типу строки результата запроса). Если возвращается несколько строк, то только первая будет присвоена переменной(ым) в INTO. Если не возвращается ни одной строки, то присваивается NULL. Без предложения INTO результаты запроса отбрасываются.

С указанием STRICT запрос должен вернуть ровно одну строку, иначе выдаётся сообщение об ошибке.

В тексте команды можно использовать значения параметров, ссылки на параметры обозначаются как $1, $2 и т. д. Эти символы указывают на значения, находящиеся в предложении USING. Такой метод зачастую предпочтительнее, чем вставка значений в команду в виде текста: он позволяет исключить во время выполнения дополнительные расходы на преобразования значений в текст и обратно, и не открывает возможности для SQL-инъекций, не требуя применять экранирование или кавычки для спецсимволов. Пример:

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

Обратите внимание, что символы параметров можно использовать только вместо значений данных. Если же требуется динамически формировать имена таблиц или столбцов, их необходимо вставлять в виде текста. Например, если в предыдущем запросе необходимо динамически задавать имя таблицы, можно сделать следующее:

```
EXECUTE 'SELECT count(*) FROM '
    || quote_ident(tabname)
    || ' WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

В качестве более аккуратного решения, вместо имени таблиц или столбцов можно использовать указание формата %I с функцией format() (текст, разделённый символами новой строки, соединяется вместе):

```
EXECUTE format('SELECT count(*) FROM %I '
   'WHERE inserted_by = $1 AND inserted <= $2', tabname)
   INTO c
   USING checked_user, checked_date;
```

Ещё одно ограничение состоит в том, что символы параметров могут использоваться только в командах SELECT, INSERT, UPDATE и DELETE. В операторы других типов (обычно называемые служебными) значения нужно вставлять в текстовом виде, даже если это просто значения данных.

Команда EXECUTE с неизменяемым текстом и параметрами USING (как в первом примере выше), функционально эквивалентна команде, записанной напрямую в PL/pgSQL, в которой переменные PL/pgSQL автоматически заменяются значениями. Важное отличие в том, что EXECUTE при каждом

исполнении заново строит план команды с учётом текущих значений параметров, тогда как PL/pgSQL строит общий план выполнения и кеширует его при повторном использовании. В тех случаях, когда наилучший план выполнения сильно зависит от значений параметров, может быть полезно использовать EXECUTE для гарантии того, что не будет выбран общий план.

В настоящее время команда SELECT INTO не поддерживается в EXECUTE, вместо этого нужно выполнять обычный SELECT и указать INTO для самой команды EXECUTE.

> **Примечание**
>
> Оператор EXECUTE в PL/pgSQL не имеет отношения к одноимённому SQL-оператору сервера Postgres Pro. Серверный EXECUTE не может напрямую использоваться в функциях на PL/pgSQL (и в этом нет необходимости).

Пример 40.1. Использование кавычек в динамических запросах

При работе с динамическими командами часто приходится иметь дело с экранированием одинарных кавычек. Рекомендуемым методом для взятия текста в кавычки в теле функции является экранирование знаками доллара. (Если имеется унаследованный код, не использующий этот метод, пожалуйста, обратитесь к обзору в Подразделе 40.11.1, это поможет сэкономить усилия при переводе кода к более приемлемому виду.)

Динамические значения требуют особого внимания, так как они могут содержать апострофы. Например, можно использовать функцию format() (предполагается, что тело функции заключается в доллары, так что апострофы дублировать не нужно):

```
EXECUTE format('UPDATE tbl SET %I = $1 '
   'WHERE key = $2', colname) USING newvalue, keyvalue;
```

Также можно напрямую вызывать функции заключения в кавычки:

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_literal(newvalue)
        || ' WHERE key = '
        || quote_literal(keyvalue);
```

Этот пример демонстрирует использование функций quote_ident и quote_literal (см. Раздел 9.4). Для надёжности, выражения, содержащие идентификаторы столбцов и таблиц должны использовать функцию quote_ident при добавлении в текст запроса. А для выражений со значениями, которые должны быть обычными строками, используется функция quote_literal. Эти функции выполняют соответствующие шаги, чтобы вернуть текст, по ситуации заключённый в двойные или одинарные кавычки и с правильно экранированными специальными символами.

Так как функция quote_literal помечена как STRICT, то она всегда возвращает NULL, если переданный ей аргумент имеет значение NULL. В приведённом выше примере, если newvalue или keyvalue были NULL, вся строка с текстом запроса станет NULL, что приведёт к ошибке в EXECUTE. Для предотвращения этой проблемы используйте функцию quote_nullable, которая работает так же, как quote_literal за исключением того, что при вызове с пустым аргументом возвращает строку 'NULL'. Например:

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_nullable(newvalue)
        || ' WHERE key = '
        || quote_nullable(keyvalue);
```

Если вы имеете дело со значениями, которые могут быть пустыми, то, как правило, нужно использовать quote_nullable вместо quote_literal.

Как обычно, необходимо убедиться, что значения NULL в запросе не принесут неожиданных результатов. Например, следующее условие WHERE

```
'WHERE key = ' || quote_nullable(keyvalue)
```

никогда не выполнится, если keyvalue — NULL, так как применение = с операндом, имеющим значение NULL, всегда даёт NULL. Если требуется, чтобы NULL обрабатывалось как обычное значение, то условие выше нужно переписать так:

```
'WHERE key IS NOT DISTINCT FROM ' || quote_nullable(keyvalue)
```

(В настоящее время IS NOT DISTINCT FROM работает менее эффективно, чем =, так что используйте этот способ, только если это действительно необходимо. Подробнее особенности NULL и IS DISTINCT описаны в Разделе 9.2.)

Обратите внимание, что использование знака $ полезно только для взятия в кавычки фиксированного текста. Плохая идея написать этот пример так:

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = $$'
        || newvalue
        || '$$ WHERE key = '
        || quote_literal(keyvalue);
```

потому что newvalue может также содержать $$. Эта же проблема может возникнуть и с любым другим разделителем, используемым после знака $. Поэтому, чтобы безопасно заключить заранее неизвестный текст в кавычки, *нужно* использовать соответствующие функции: quote_literal, quote_nullable, или quote_ident.

Динамические операторы SQL также можно безопасно сформировать, используя функцию format (см. Раздел 9.4). Например:

```
EXECUTE format('UPDATE tbl SET %I = %L '
   'WHERE key = %L', colname, newvalue, keyvalue);
```

Указание %I равнозначно вызову quote_ident, а %L — вызову quote_nullable. Функция format может применяться в сочетании с предложением USING:

```
EXECUTE format('UPDATE tbl SET %I = $1 WHERE key = $2', colname)
   USING newvalue, keyvalue;
```

Эта форма лучше, так как с ней переменные обрабатываются в их собственном формате данных, а не преобразуются безусловно в текст, чтобы затем выводиться с использованием %L. Она также и более эффективна.

Более объёмный пример использования динамической команды и EXECUTE можно увидеть в Примере 40.10. В нём создаётся и динамически выполняется команда CREATE FUNCTION для определения новой функции.

40.5.5. Статус выполнения команды

Определить результат команды можно несколькими способами. Во-первых, можно воспользоваться командой GET DIAGNOSTICS, имеющей форму:

```
GET [ CURRENT ] DIAGNOSTICS переменная { = | := } элемент [ , ... ];
```

Эта команда позволяет получить системные индикаторы состояния. Слово CURRENT не несёт смысловой нагрузки (но см. также описание GET STACKED DIAGNOSTICS в Подразделе 40.6.6.1). Каждый *элемент* представляется ключевым словом, указывающим, какое значение состояния нужно присвоить заданной *переменной* (она должна иметь подходящий тип данных, чтобы принять

его). Доступные в настоящее время элементы состояния показаны в Таблице 40.1. Вместо принятого в стандарте SQL присваивания (=) можно применять присваивание с двоеточием (:=). Например:

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

Таблица 40.1. Доступные элементы диагностики

Имя	Тип	Описание
ROW_COUNT	bigint	число строк, обработанных последней командой SQL
RESULT_OID	oid	OID последней строки, вставленной предыдущей командой SQL (полезен только после команды INSERT для таблицы, содержащей OID)
PG_CONTEXT	text	строки текста, описывающие текущий стек вызовов (см. Подраздел 40.6.7)

Второй способ определения статуса выполнения команды заключается в проверке значения специальной переменной FOUND, имеющей тип boolean. При вызове функции на PL/pgSQL, переменная FOUND инициализируется в ложь. Далее, значение переменной изменяется следующими операторами:

- SELECT INTO записывает в FOUND true, если строка присвоена, или false, если строки не были получены.
- PERFORM записывает в FOUND true, если строки выбраны (и отброшены) или false, если строки не выбраны.
- UPDATE, INSERT и DELETE записывают в FOUND true, если при их выполнении была задействована хотя бы одна строка, или false, если ни одна строка не была задействована.
- FETCH записывают в FOUND true, если команда вернула строку, или false, если строка не выбрана.
- MOVE записывают в FOUND true при успешном перемещении курсора, в противном случае — false.
- FOR, как и FOREACH, записывает в FOUND true, если была произведена хотя бы одна итерация цикла, в противном случае — false. При этом значение FOUND будет установлено только после выхода из цикла. Пока цикл выполняется, оператор цикла не изменяет значение переменной. Но другие операторы внутри цикла могут менять значение FOUND.
- RETURN QUERY и RETURN QUERY EXECUTE записывают в FOUND true, если запрос вернул хотя бы одну строку, или false, если строки не выбраны.

Другие операторы PL/pgSQL не меняют значение FOUND. Помните в частности, что EXECUTE изменяет вывод GET DIAGNOSTICS, но не меняет FOUND.

FOUND является локальной переменной в каждой функции PL/pgSQL и любые её изменения, влияют только на текущую функцию.

40.5.6. Не делать ничего

Иногда бывает полезен оператор, который не делает ничего. Например, он может показывать, что одна из ветвей if/then/else сознательно оставлена пустой. Для этих целей используется NULL:

```
NULL;
```

В следующем примере два фрагмента кода эквивалентны:

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        NULL;  -- ошибка игнорируется
END;
```

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN  -- ошибка игнорируется
END;
```

Какой вариант выбрать — дело вкуса.

> **Примечание**
>
> В Oracle PL/SQL не допускаются пустые списки операторов, поэтому `NULL` *обязателен* в подобных ситуациях. В PL/pgSQL разрешается не писать ничего.

40.6. Управляющие структуры

Управляющие структуры, вероятно, наиболее полезная и важная часть PL/pgSQL. С их помощью можно очень гибко и эффективно манипулировать данными Postgres Pro.

40.6.1. Команды для возврата значения из функции

Две команды позволяют вернуть данные из функции: `RETURN` и `RETURN NEXT`.

40.6.1.1. `RETURN`

```
RETURN выражение;
```

`RETURN` с последующим выражением прекращает выполнение функции и возвращает значение выражения в вызывающую программу. Эта форма используется для функций PL/pgSQL, которые не возвращают набор строк.

В функции, возвращающей скалярный тип, результирующее выражение автоматически приводится к типу возвращаемого значения. Однако если возвращаемый тип —составной (строка), возвращаемое выражение должно в точности содержать требуемый набор столбцов. При этом может потребоваться явное приведение типов.

Для функции с выходными параметрами просто используйте `RETURN` без выражения. Будут возвращены текущие значения выходных параметров.

Для функции, возвращающей `void`, `RETURN` можно использовать в любом месте, но без выражения после `RETURN`.

Возвращаемое значение функции не может остаться не определённым. Если достигнут конец блока верхнего уровня, а оператор `RETURN` так и не встретился, происходит ошибка времени выполнения. Это не касается функций с выходными параметрами и функций, возвращающих `void`. Для них оператор `RETURN` выполняется автоматически по окончании блока верхнего уровня.

Несколько примеров:

```
-- Функции, возвращающие скалярный тип данных
RETURN 1 + 2;
RETURN scalar_var;
```

```
-- Функции, возвращающие составной тип данных
RETURN composite_type_var;
RETURN (1, 2, 'three'::text);  -- требуется приведение типов
```

40.6.1.2. RETURN NEXT и RETURN QUERY

```
RETURN NEXT выражение;
RETURN QUERY запрос;
RETURN QUERY EXECUTE строка-команды [USING выражение [, ...]];
```

Для функций на PL/pgSQL, возвращающих `SETOF некий_тип`, нужно действовать несколько по-иному. Отдельные элементы возвращаемого значения формируются командами `RETURN NEXT` или `RETURN QUERY`, а финальная команда `RETURN` без аргументов завершает выполнение функции. `RETURN NEXT` используется как со скалярными, так и с составными типами данных. Для составного типа результат функции возвращается в виде таблицы. `RETURN QUERY` добавляет результат выполнения запроса к результату функции. `RETURN NEXT` и `RETURN QUERY` можно свободно смешивать в теле функции, в этом случае их результаты будут объединены.

`RETURN NEXT` и `RETURN QUERY` не выполняют возврат из функции. Они просто добавляют строки в результирующее множество. Затем выполнение продолжается со следующего оператора в функции. Успешное выполнение `RETURN NEXT` и `RETURN QUERY` формирует множество строк результата. Для выхода из функции используется `RETURN`, обязательно без аргументов (или можно просто дождаться окончания выполнения функции).

`RETURN QUERY` имеет разновидность `RETURN QUERY EXECUTE`, предназначенную для динамического выполнения запроса. В текст запроса можно добавить параметры, используя `USING`, так же как и с командой `EXECUTE`.

Для функции с выходными параметрами просто используйте `RETURN NEXT` без аргументов. При каждом исполнении `RETURN NEXT` текущие значения выходных параметров сохраняются для последующего возврата в качестве строки результата. Обратите внимание, что если функция с выходными параметрами должна возвращать множество значений, то при объявлении нужно указывать `RETURNS SETOF`. При этом если выходных параметров несколько, то используется `RETURNS SETOF record`, а если только один с типом *некий_тип*, то `RETURNS SETOF некий_тип`.

Пример использования `RETURN NEXT`:

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION get_all_foo() RETURNS SETOF foo AS
$BODY$
DECLARE
    r foo%rowtype;
BEGIN
    FOR r IN
        SELECT * FROM foo WHERE fooid > 0
    LOOP
        -- здесь возможна обработка данных
        RETURN NEXT r; -- возвращается текущая строка запроса
    END LOOP;
    RETURN;
END;
$BODY$
LANGUAGE plpgsql;

SELECT * FROM get_all_foo();
```

Пример использования `RETURN QUERY`:

```
CREATE FUNCTION get_available_flightid(date) RETURNS SETOF integer AS
$BODY$
BEGIN
    RETURN QUERY SELECT flightid
                   FROM flight
                  WHERE flightdate >= $1
                    AND flightdate < ($1 + 1);

    -- Так как выполнение ещё не закончено, можно проверить, были ли возвращены строки,
    -- и выдать исключение, если нет.
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Нет рейсов на дату: %.', $1;
    END IF;

    RETURN;
 END;
$BODY$
LANGUAGE plpgsql;

-- Возвращает доступные рейсы либо вызывает исключение, если таковых нет.
SELECT * FROM get_available_flightid(CURRENT_DATE);
```

> **Примечание**
>
> В текущей реализации `RETURN NEXT` и `RETURN QUERY` результирующее множество накапливается целиком, прежде чем будет возвращено из функции. Если множество очень большое, то это может отрицательно сказаться на производительности, так как при нехватке оперативной памяти данные записываются на диск. В следующих версиях PL/pgSQL это ограничение будет снято. В настоящее время управлять количеством оперативной памяти в подобных случаях можно параметром конфигурации work_mem. При наличии свободной памяти администраторы должны рассмотреть возможность увеличения значения данного параметра.

40.6.2. Условные операторы

Операторы `IF` и `CASE` позволяют выполнять команды в зависимости от определённых условий. PL/pgSQL поддерживает три формы `IF`:

- `IF ... THEN ... END IF`
- `IF ... THEN ... ELSE ... END IF`
- `IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF`

и две формы `CASE`:

- `CASE ... WHEN ... THEN ... ELSE ... END CASE`
- `CASE WHEN ... THEN ... ELSE ... END CASE`

40.6.2.1. `IF-THEN`

```
IF логическое-выражение THEN
    операторы
END IF;
```

`IF-THEN` это простейшая форма `IF`. Операторы между `THEN` и `END IF` выполняются, если условие (*логическое-выражение*) истинно. В противном случае они опускаются.

Пример:

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

40.6.2.2. IF-THEN-ELSE

```
IF логическое-выражение THEN
    операторы
ELSE
    операторы
END IF;
```

IF-THEN-ELSE добавляет к IF-THEN возможность указать альтернативный набор операторов, которые будут выполнены, если условие не истинно (в том числе, если условие NULL).

Примеры:

```
IF parentid IS NULL OR parentid = ''
THEN
    RETURN fullname;
ELSE
    RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;
```

```
IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

40.6.2.3. IF-THEN-ELSIF

```
IF логическое-выражение THEN
    операторы
[ELSIF логическое-выражение THEN операторы [ELSIF логическое-выражение THEN операторы
 ...]]
[ELSE операторы]
END IF;
```

В некоторых случаях двух альтернатив недостаточно. IF-THEN-ELSIF обеспечивает удобный способ проверки нескольких вариантов по очереди. Условия в IF последовательно проверяются до тех пор, пока не будет найдено первое истинное. После этого операторы, относящиеся к этому условию, выполняются, и управление переходит к следующей после END IF команде. (Все последующие условия не проверяются.) Если ни одно из условий IF не является истинным, то выполняется блок ELSE (если присутствует).

Пример:

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- остаётся только один вариант: number имеет значение NULL
    result := 'NULL';
END IF;
```

Вместо ключевого слова ELSIF можно использовать ELSEIF.

Другой вариант сделать то же самое, это использование вложенных операторов IF-THEN-ELSE, как в следующем примере:

```
IF demo_row.sex = 'm' THEN
    pretty_sex := 'man';
ELSE
    IF demo_row.sex = 'f' THEN
        pretty_sex := 'woman';
    END IF;
END IF;
```

Однако это требует написания соответствующих END IF для каждого IF, что при наличии нескольких альтернатив делает код более громоздким, чем использование ELSIF.

40.6.2.4. Простой CASE

```
CASE выражение-поиска
    WHEN выражение [, выражение [...]] THEN
      операторы
  [WHEN выражение [, выражение [...]] THEN операторы ...]
  [ELSE операторы]
END CASE;
```

Простая форма CASE реализует условное выполнение на основе сравнения операндов. *Выражение-поиска* вычисляется (один раз) и последовательно сравнивается с каждым *выражением* в условиях WHEN. Если совпадение найдено, то выполняются соответствующие *операторы* и управление переходит к следующей после END CASE команде. (Все последующие выражения WHEN не проверяются.) Если совпадение не было найдено, то выполняются *операторы* в ELSE. Но если ELSE нет, то вызывается исключение CASE_NOT_FOUND.

Пример:

```
CASE x
    WHEN 1, 2 THEN
        msg := 'один или два';
    ELSE
        msg := 'значение, отличное от один или два';
END CASE;
```

40.6.2.5. CASE с перебором условий

```
CASE
    WHEN логическое-выражение THEN
      операторы
  [WHEN логическое-выражение THEN операторы ...]
  [ELSE операторы]
END CASE;
```

Эта форма CASE реализует условное выполнение, основываясь на истинности логических условий. Каждое *логическое-выражение* в предложении WHEN вычисляется по порядку до тех пор, пока не будет найдено истинное. Затем выполняются соответствующие *операторы* и управление переходит к следующей после END CASE команде. (Все последующие выражения WHEN не проверяются.) Если ни одно из условий не окажется истинным, то выполняются *операторы* в ELSE. Но если ELSE нет, то вызывается исключение CASE_NOT_FOUND.

Пример:

```
CASE
    WHEN x BETWEEN 0 AND 10 THEN
        msg := 'значение в диапазоне между 0 и 10';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'значение в диапазоне между 11 и 20';
```

```
END CASE;
```

Эта форма CASE полностью эквивалента IF-THEN-ELSIF, за исключением того, что при невыполнении всех условий и отсутствии ELSE, IF-THEN-ELSIF ничего не делает, а CASE вызывает ошибку.

40.6.3. Простые циклы

Операторы LOOP, EXIT, CONTINUE, WHILE, FOR и FOREACH позволяют повторить серию команд в функции на PL/pgSQL.

40.6.3.1. LOOP

```
[<<метка>>]
LOOP
    операторы
END LOOP [ метка ];
```

LOOP организует безусловный цикл, который повторяется до бесконечности, пока не будет прекращён операторами EXIT или RETURN. Для вложенных циклов можно использовать *метку* в операторах EXIT и CONTINUE, чтобы указать, к какому циклу эти операторы относятся.

40.6.3.2. EXIT

```
EXIT [ метка ] [WHEN логическое-выражение];
```

Если *метка* не указана, то завершается самый внутренний цикл, далее выполняется оператор, следующий за END LOOP. Если *метка* указана, то она должна относиться к текущему или внешнему циклу, или это может быть метка блока. При этом в именованном цикле/блоке выполнение прекращается, а управление переходит к следующему оператору после соответствующего END.

При наличии WHEN цикл прекращается, только если *логическое-выражение* истинно. В противном случае управление переходит к оператору, следующему за EXIT.

EXIT можно использовать со всеми типами циклов, не только с безусловным.

Когда EXIT используется для выхода из блока, управление переходит к следующему оператору после окончания блока. Обратите внимание, что для выхода из блока нужно обязательно указывать *метку*. EXIT без *метки* не позволяет прекратить работу блока. (Это изменение по сравнению с версиями PostgreSQL до 8.4, в которых разрешалось использовать EXIT без *метки* для прекращения работы текущего блока.)

Примеры:

```
LOOP
    -- здесь производятся вычисления
    IF count > 0 THEN
        EXIT;  -- выход из цикла
    END IF;
END LOOP;

LOOP
    -- здесь производятся вычисления
    EXIT WHEN count > 0;  -- аналогично предыдущему примеру
END LOOP;

<<ablock>>
BEGIN
    -- здесь производятся вычисления
    IF stocks > 100000 THEN
        EXIT ablock;  -- выход из блока BEGIN
```

```
    END IF;
    -- вычисления не будут выполнены, если stocks > 100000
END;
```

40.6.3.3. CONTINUE

```
CONTINUE [ метка ] [WHEN логическое-выражение];
```

Если *метка* не указана, то начинается следующая итерация самого внутреннего цикла. То есть все оставшиеся в цикле операторы пропускаются, и управление переходит к управляющему выражению цикла (если есть) для определения, нужна ли ещё одна итерация цикла. Если *метка* присутствует, то она указывает на метку цикла, выполнение которого будет продолжено.

При наличии WHEN следующая итерация цикла начинается только тогда, когда *логическое-выражение* истинно. В противном случае управление переходит к оператору, следующему за CONTINUE.

CONTINUE можно использовать со всеми типами циклов, не только с безусловным.

Примеры:

```
LOOP
    -- здесь производятся вычисления
    EXIT WHEN count > 100;
    CONTINUE WHEN count < 50;
    -- вычисления для count в диапазоне 50 .. 100
END LOOP;
```

40.6.3.4. WHILE

```
[<<метка>>]
WHILE логическое-выражение LOOP
    операторы
END LOOP [ метка ];
```

WHILE выполняет серию команд до тех пор, пока истинно *логическое-выражение*. Выражение проверяется непосредственно перед каждым входом в тело цикла.

Пример:

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- здесь производятся вычисления
END LOOP;

WHILE NOT done LOOP
    -- здесь производятся вычисления
END LOOP;
```

40.6.3.5. FOR (целочисленный вариант)

```
[<<метка>>]
FOR имя IN [REVERSE] выражение .. выражение [BY выражение] LOOP
    операторы
END LOOP [ метка ];
```

В этой форме цикла FOR итерации выполняются по диапазону целых чисел. Переменная *имя* автоматически определяется с типом integer и существует только внутри цикла (если уже существует переменная с таким именем, то внутри цикла она будет игнорироваться). Выражения для нижней и верхней границы диапазона чисел вычисляются один раз при входе в цикл. Если не указано BY, то шаг итерации 1, в противном случае используется значение в BY, которое вычисляется, опять же, один раз при входе в цикл. Если указано REVERSE, то после каждой итерации величина шага вычитается, а не добавляется.

Примеры целочисленного FOR:

```
FOR i IN 1..10 LOOP
    -- внутри цикла переменная i будет иметь значения 1,2,3,4,5,6,7,8,9,10
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- внутри цикла переменная i будет иметь значения 10,9,8,7,6,5,4,3,2,1
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
    -- внутри цикла переменная i будет иметь значения 10,8,6,4,2
END LOOP;
```

Если нижняя граница цикла больше верхней границы (или меньше, в случае REVERSE), то тело цикла не выполняется вообще. При этом ошибка не возникает.

Если с циклом FOR связана *метка*, к целочисленной переменной цикла можно обращаться по имени, указывая эту *метку*.

40.6.4. Цикл по результатам запроса

Другой вариант FOR позволяет организовать цикл по результатам запроса. Синтаксис:

```
[ <<метка>> ]
FOR цель IN запрос LOOP
    операторы
END LOOP [ метка ];
```

Переменная *цель* может быть переменной-кортежем, переменной типа record или разделённым запятыми списком скалярных переменных. Переменной *цель* последовательно присваиваются строки результата запроса, и для каждой строки выполняется тело цикла. Пример:

```
CREATE FUNCTION refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Refreshing all materialized views...';

    FOR mviews IN
       SELECT n.nspname AS mv_schema,
              c.relname AS mv_name,
              pg_catalog.pg_get_userbyid(c.relowner) AS owner
         FROM pg_catalog.pg_class c
    LEFT JOIN pg_catalog.pg_namespace n ON (n.oid = c.relnamespace)
        WHERE c.relkind = 'm'
     ORDER BY 1
    LOOP

        -- Здесь "mviews" содержит одну запись с информацией о матпредставлении

        RAISE NOTICE 'Refreshing materialized view %.% (owner: %)...',
                     quote_ident(mviews.mv_schema),
                     quote_ident(mviews.mv_name),
                     quote_ident(mviews.owner);
        EXECUTE format('REFRESH MATERIALIZED VIEW %I.%I', mviews.mv_schema,
 mviews.mv_name);
    END LOOP;

    RAISE NOTICE 'Done refreshing materialized views.';
    RETURN 1;
```

```
END;
$$ LANGUAGE plpgsql;
```

Если цикл завершается по команде EXIT, то последняя присвоенная строка доступна и после цикла.

В качестве *запроса* в этом типе оператора FOR может задаваться любая команда SQL, возвращающая строки. Чаще всего это SELECT, но также можно использовать и INSERT, UPDATE или DELETE с предложением RETURNING. Кроме того, возможно применение и некоторых служебных команд, например EXPLAIN.

Для переменных PL/pgSQL в тексте запроса выполняется подстановка значений, план запроса кешируется для возможного повторного использования, как подробно описано в Подразделе 40.10.1 и Подразделе 40.10.2.

Ещё одна разновидность этого типа цикла FOR-IN-EXECUTE:

```
[ <<метка>> ]
FOR цель IN EXECUTE выражение_проверки [ USING выражение [, ... ] ] LOOP
    операторы
END LOOP [ метка ];
```

Она похожа на предыдущую форму, за исключением того, что текст запроса указывается в виде строкового выражения. Текст запроса формируется и для него строится план выполнения при каждом входе в цикл. Это даёт программисту выбор между скоростью предварительно разобранного запроса и гибкостью динамического запроса, так же, как и в случае с обычным оператором EXECUTE. Как и в EXECUTE, значения параметров могут быть добавлены в команду с использованием USING.

Ещё один способ организовать цикл по результатам запроса это объявить курсор. Описание в Подразделе 40.7.4.

40.6.5. Цикл по элементам массива

Цикл FOREACH очень похож на FOR. Отличие в том, что вместо перебора строк SQL-запроса происходит перебор элементов массива. (В целом, FOREACH предназначен для перебора выражений составного типа. Варианты реализации цикла для работы с прочими составными выражениями помимо массивов могут быть добавлены в будущем.) Синтаксис цикла FOREACH:

```
[ <<метка>> ]
FOREACH цель [ SLICE число ] IN ARRAY выражение LOOP
    операторы
END LOOP [ метка ];
```

Без указания SLICE, или если SLICE равен 0, цикл выполняется по всем элементам массива, полученного из *выражения*. Переменной *цель* последовательно присваивается каждый элемент массива и для него выполняется тело цикла. Пример цикла по элементам целочисленного массива:

```
CREATE FUNCTION sum(int[]) RETURNS int8 AS $$
DECLARE
  s int8 := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY $1
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

Обход элементов проводится в том порядке, в котором они сохранялись, независимо от размерности массива. Как правило, *цель* это одиночная переменная, но может быть и списком

переменных, когда элементы массива имеют составной тип (записи). В этом случае переменным присваиваются значения из последовательных столбцов составного элемента массива.

При положительном значении SLICE FOREACH выполняет итерации по срезам массива, а не по отдельным элементам. Значение SLICE должно быть целым числом, не превышающим размерности массива. Переменная *цель* должна быть массивом, который получает последовательные срезы исходного массива, где размерность каждого среза задаётся значением SLICE. Пример цикла по одномерным срезам:

```
CREATE FUNCTION scan_rows(int[]) RETURNS void AS $$
DECLARE
  x int[];
BEGIN
  FOREACH x SLICE 1 IN ARRAY $1
  LOOP
    RAISE NOTICE 'row = %', x;
  END LOOP;
END;
$$ LANGUAGE plpgsql;

SELECT scan_rows(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);

NOTICE:  row = {1,2,3}
NOTICE:  row = {4,5,6}
NOTICE:  row = {7,8,9}
NOTICE:  row = {10,11,12}
```

40.6.6. Обработка ошибок

По умолчанию любая возникающая ошибка прерывает выполнение функции на PL/pgSQL и транзакцию, в которая она выполняется. Использование в блоке секции EXCEPTION позволяет перехватывать и обрабатывать ошибки. Синтаксис секции EXCEPTION расширяет синтаксис обычного блока:

```
[ <<метка>> ]
[ DECLARE
    объявления ]
BEGIN
    операторы
EXCEPTION
    WHEN условие [ OR условие ... ] THEN
        операторы_обработчика
    [ WHEN условие [ OR условие ... ] THEN
          операторы_обработчика
      ... ]
END;
```

Если ошибок не было, то выполняются все *операторы* блока и управление переходит к следующему оператору после END. Но если при выполнении *оператора* происходит ошибка, то дальнейшая обработка прекращается и управление переходит к списку исключений в секции EXCEPTION. В этом списке ищется первое исключение, условие которого соответствует ошибке. Если исключение найдено, то выполняются соответствующие *операторы_обработчика* и управление переходит к следующему оператору после END. Если исключение не найдено, то ошибка передаётся наружу, как будто секции EXCEPTION не было. При этом ошибку можно перехватить в секции EXCEPTION внешнего блока. Если ошибка так и не была перехвачена, то обработка функции прекращается.

В качестве *условия* может задаваться одно из имён, перечисленных в Приложении А. Если задаётся имя категории, ему соответствуют все ошибки в данной категории. Специальному имени условия OTHERS (другие) соответствуют все типы ошибок, кроме QUERY_CANCELED и ASSERT_FAILURE. (И эти два типа ошибок можно перехватить по имени, но часто это неразумно.) Имена условий

воспринимаются без учёта регистра. Условие ошибки также можно задать кодом SQLSTATE; например, эти два варианта равнозначны:

```
WHEN division_by_zero THEN ...
WHEN SQLSTATE '22012' THEN ...
```

Если при выполнении *операторов_обработчика* возникнет новая ошибка, то она не может быть перехвачена в этой секции EXCEPTION. Ошибка передаётся наружу и её можно перехватить в секции EXCEPTION внешнего блока.

При выполнении команд в секции EXCEPTION локальные переменные функции на PL/pgSQL сохраняют те значения, которые были на момент возникновения ошибки. Однако все изменения в базе данных, выполненные в блоке, будут отменены. В качестве примера рассмотрим следующий фрагмент:

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
    UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'перехватили ошибку division_by_zero';
        RETURN x;
END;
```

При присваивании значения переменной y произойдёт ошибка division_by_zero. Она будет перехвачена в секции EXCEPTION. Оператор RETURN вернёт значение x, увеличенное на единицу, но изменения сделанные командой UPDATE будут отменены. Изменения, выполненные командой INSERT, которая предшествует блоку, не будут отменены. В результате, база данных будет содержать Tom Jones, а не Joe Jones.

> **Подсказка**
>
> Наличие секции EXCEPTION значительно увеличивает накладные расходы на вход/выход из блока, поэтому не используйте EXCEPTION без надобности.

Пример 40.2. Обработка исключений для команд UPDATE/INSERT

В этом примере обработка исключений помогает выполнить либо команду UPDATE, либо INSERT, в зависимости от ситуации. Однако в современных приложениях вместо этого приёма рекомендуется использовать INSERT с ON CONFLICT DO UPDATE. Данный пример предназначен в первую очередь для демонстрации управления выполнением PL/pgSQL:

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
BEGIN
    LOOP
        -- сначала попытаться изменить запись по ключу
        UPDATE db SET b = data WHERE a = key;
        IF found THEN
            RETURN;
        END IF;
        -- записи с таким ключом нет, поэтому её нужно добавить
        -- если параллельно будет вставлена запись с таким же ключом,
        -- произойдёт ошибка уникальности
        BEGIN
```

```
            INSERT INTO db(a,b) VALUES (key, data);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- здесь не нужно ничего делать,
            -- просто продолжить цикл, чтобы повторить UPDATE.
        END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```

В этом коде предполагается, что ошибка `unique_violation` вызывается самой командой `INSERT`, а не, скажем, внутренним оператором `INSERT` в функции триггера для этой таблицы. Некорректное поведение также возможно, если в таблице будет несколько уникальных индексов; тогда операция будет повторяться вне зависимости от того, нарушение какого индекса вызвало ошибку. Используя средства, рассмотренные далее, можно сделать код более надёжным, проверяя, что перехвачена именно ожидаемая ошибка.

40.6.6.1. Получение информации об ошибке

При обработке исключений часто бывает необходимым получить детальную информацию о произошедшей ошибке. Для этого в PL/pgSQL есть два способа: использование специальных переменных и команда `GET STACKED DIAGNOSTICS`.

Внутри секции `EXCEPTION` специальная переменная `SQLSTATE` содержит код ошибки, для которой было вызвано исключение (список возможных кодов ошибок приведён в Таблице A.1). Специальная переменная `SQLERRM` содержит сообщение об ошибке, связанное с исключением. Эти переменные являются неопределёнными вне секции `EXCEPTION`.

Также в обработчике исключения можно получить информацию о текущем исключении командой `GET STACKED DIAGNOSTICS`, которая имеет вид:

```
GET STACKED DIAGNOSTICS переменная { = | := } элемент [ , ... ];
```

Каждый *`элемент`* представляется ключевым словом, указывающим, какое значение состояния нужно присвоить заданной *`переменной`* (она должна иметь подходящий тип данных, чтобы принять его). Доступные в настоящее время элементы состояния показаны в Таблице 40.2.

Таблица 40.2. Элементы диагностики ошибок

Имя	Тип	Описание
RETURNED_SQLSTATE	text	код исключения, возвращаемый SQLSTATE
COLUMN_NAME	text	имя столбца, относящегося к исключению
CONSTRAINT_NAME	text	имя ограничения целостности, относящегося к исключению
PG_DATATYPE_NAME	text	имя типа данных, относящегося к исключению
MESSAGE_TEXT	text	текст основного сообщения исключения
TABLE_NAME	text	имя таблицы, относящейся к исключению
SCHEMA_NAME	text	имя схемы, относящейся к исключению

Имя	Тип	Описание
PG_EXCEPTION_DETAIL	text	текст детального сообщения исключения (если есть)
PG_EXCEPTION_HINT	text	текст подсказки к исключению (если есть)
PG_EXCEPTION_CONTEXT	text	строки текста, описывающие стек вызовов в момент исключения (см. Подраздел 40.6.7)

Если исключение не устанавливает значение для идентификатора, то возвращается пустая строка.

Пример:

```
DECLARE
  text_var1 text;
  text_var2 text;
  text_var3 text;
BEGIN
  -- здесь происходит обработка, которая может вызвать исключение
  ...
EXCEPTION WHEN OTHERS THEN
  GET STACKED DIAGNOSTICS text_var1 = MESSAGE_TEXT,
                          text_var2 = PG_EXCEPTION_DETAIL,
                          text_var3 = PG_EXCEPTION_HINT;
END;
```

40.6.7. Получение информации о месте выполнения

Команда GET DIAGNOSTICS, ранее описанная в Подразделе 40.5.5, получает информацию о текущем состоянии выполнения кода (тогда как команда GET STACKED DIAGNOSTICS, рассмотренная ранее, выдаёт информацию о состоянии выполнения в момент предыдущей ошибки). Её элемент состояния PG_CONTEXT позволяет определить текущее место выполнения кода. PG_CONTEXT возвращает текст с несколькими строками, описывающий стек вызова. В первой строке отмечается текущая функция и выполняемая в данный момент команда GET DIAGNOSTICS, а во второй и последующих строках отмечаются функции выше по стеку вызовов. Например:

```
CREATE OR REPLACE FUNCTION outer_func() RETURNS integer AS $$
BEGIN
  RETURN inner_func();
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION inner_func() RETURNS integer AS $$
DECLARE
  stack text;
BEGIN
  GET DIAGNOSTICS stack = PG_CONTEXT;
  RAISE NOTICE E'--- Стек вызова ---\n%', stack;
  RETURN 1;
END;
$$ LANGUAGE plpgsql;

SELECT outer_func();

NOTICE:  --- Стек вызова ---
PL/pgSQL function inner_func() line 5 at GET DIAGNOSTICS
PL/pgSQL function outer_func() line 3 at RETURN
```

```
CONTEXT:  PL/pgSQL function outer_func() line 3 at RETURN
 outer_func
 ------------
           1
(1 row)
```

`GET STACKED DIAGNOSTICS ... PG_EXCEPTION_CONTEXT` возвращает похожий стек вызовов, но описывает не текущее место, а место, в котором произошла ошибка.

40.7. Курсоры

Вместо того чтобы сразу выполнять весь запрос, есть возможность настроить курсор, инкапсулирующий запрос, и затем получать результат запроса по нескольку строк за раз. Одна из причин так делать заключается в том, чтобы избежать переполнения памяти, когда результат содержит большое количество строк. (Пользователям PL/pgSQL не нужно об этом беспокоиться, так как циклы `FOR` автоматически используют курсоры, чтобы избежать проблем с памятью.) Более интересным вариантом использования является возврат из функции ссылки на курсор, что позволяет вызывающему получать строки запроса. Это эффективный способ получать большие наборы строк из функций.

40.7.1. Объявление курсорных переменных

Доступ к курсорам в PL/pgSQL осуществляется через курсорные переменные, которые всегда имеют специальный тип данных `refcursor`. Один из способов создать курсорную переменную, просто объявить её как переменную типа `refcursor`. Другой способ заключается в использовании синтаксиса объявления курсора, который в общем виде выглядит так:

имя [[NO] SCROLL] CURSOR [(*аргументы*)] FOR *запрос*;

(Для совместимости с Oracle, `FOR` можно заменять на `IS`.) С указанием `SCROLL` курсор можно будет прокручивать назад. При `NO SCROLL` прокрутка назад не разрешается. Если ничего не указано, то возможность прокрутки назад зависит от запроса. Если указаны *аргументы*, то они должны представлять собой пары *имя тип_данных*, разделённые через запятую. Эти пары определяют имена, которые будут заменены значениями параметров в данном запросе. Фактические значения для замены этих имён появятся позже, при открытии курсора.

Примеры:

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) FOR SELECT * FROM tenk1 WHERE unique1 = key;
```

Все три переменные имеют тип данных `refcursor`. Первая может быть использована с любым запросом, вторая связана (`bound`) с полностью сформированным запросом, а последняя связана с параметризованным запросом. (`key` будет заменён целочисленным значением параметра при открытии курсора.) Про переменную `curs1` говорят, что она является несвязанной (`unbound`), так как к ней не привязан никакой запрос.

40.7.2. Открытие курсора

Прежде чем получать строки из курсора, его нужно открыть. (Это эквивалентно действию SQL-команды `DECLARE CURSOR`.) В PL/pgSQL есть три формы оператора `OPEN`, две из которых используются для несвязанных курсорных переменных, а третья для связанных.

> **Примечание**
>
> Связанные курсорные переменные можно использовать с циклом `FOR` без явного открытия курсора, как описано в Подразделе 40.7.4.

40.7.2.1. OPEN FOR *запрос*

```
OPEN несвязанная_переменная_курсора [[NO] SCROLL] FOR запрос;
```

Курсорная переменная открывается и получает конкретный запрос для выполнения. Курсор не может уже быть открытым, а курсорная переменная обязана быть несвязанной (то есть просто переменной типа `refcursor`). Запрос должен быть командой `SELECT` или любой другой, которая возвращает строки (к примеру `EXPLAIN`). Запрос обрабатывается так же, как и другие команды SQL в PL/pgSQL: имена переменных PL/pgSQL заменяются на значения, план запроса кешируется для повторного использования. Подстановка значений переменных PL/pgSQL проводится при открытии курсора командой `OPEN`, последующие изменения значений переменных не влияют на работу курсора. `SCROLL` и `NO SCROLL` имеют тот же смысл, что и для связанного курсора.

Пример:

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

40.7.2.2. OPEN FOR EXECUTE

```
OPEN несвязанная_переменная_курсора [[NO] SCROLL] FOR EXECUTE строка_запроса
                                    [USING выражение [, ...]];
```

Переменная курсора открывается и получает конкретный запрос для выполнения. Курсор не может быть уже открыт и он должен быть объявлен как несвязанная переменная курсора (то есть, как просто переменная `refcursor`). Запрос задаётся строковым выражением, так же, как в команде `EXECUTE`. Как обычно, это даёт возможность гибко менять план запроса от раза к разу (см. Подраздел 40.10.2). Это также означает, что замена переменных происходит не в самой строке команды. Как и с `EXECUTE`, значения параметров вставляются в динамическую команду, используя `format()` и `USING`. Параметры `SCROLL` и `NO SCROLL` здесь действуют так же, как и со связанным курсором.

Пример:

```
OPEN curs1 FOR EXECUTE format('SELECT * FROM %I WHERE col1 = $1',tabname) USING
 keyvalue;
```

В этом примере в текст запроса вставляется имя таблицы с применением `format()`. Значение, сравниваемое с `col1`, вставляется посредством параметра `USING`, так что заключать его в апострофы не нужно.

40.7.2.3. Открытие связанного курсора

```
OPEN связанная_переменная_курсора [( [имя_аргумента :=] значение_аргумента [, ...] )];
```

Эта форма `OPEN` используется для открытия курсорной переменной, которая была связана с запросом при объявлении. Курсор не может уже быть открытым. Список фактических значений аргументов должен присутствовать только в том случае, если курсор объявлялся с параметрами. Эти значения будут подставлены в запрос.

План запроса для связанного курсора всегда считается кешируемым. В этом случае нет эквивалента `EXECUTE`. Обратите внимание, что `SCROLL` и `NO SCROLL` не могут быть указаны в этой форме `OPEN`, возможность прокрутки назад была определена при объявлении курсора.

При передаче значений аргументов можно использовать позиционную или именную нотацию. В позиционной нотации все аргументы указываются по порядку. В именной нотации имя каждого аргумента отделяется от выражения аргумента с помощью `:=`. Это подобно вызову функций, описанному в Разделе 4.3. Также разрешается смешивать позиционную и именную нотации.

Примеры (здесь используются ранее объявленные курсоры):

```
OPEN curs2;
OPEN curs3(42);
OPEN curs3(key := 42);
```

Так как для связанного курсора выполняется подстановка значений переменных, то, на самом деле, существует два способа передать значения в курсор. Либо использовать явные аргументы в OPEN, либо неявно, ссылаясь на переменные PL/pgSQL в запросе. В связанном курсоре можно ссылаться только на те переменные, которые были объявлены до самого курсора. В любом случае значение переменной для подстановки в запрос будет определяться на момент выполнения OPEN. Вот ещё один способ получить тот же результат с curs3, как в примере выше:

```
DECLARE
    key integer;
    curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
BEGIN
    key := 42;
    OPEN curs4;
```

40.7.3. Использование курсоров

После того как курсор будет открыт, с ним можно работать при помощи описанных здесь операторов.

Работать с курсором необязательно в той же функции, где он был открыт. Из функции можно вернуть значение с типом refcursor, что позволит вызывающему продолжить работу с курсором. (Внутри refcursor представляет собой обычное строковое имя так называемого портала, содержащего активный запрос курсора. Это имя можно передавать, присваивать другим переменным с типом refcursor и так далее, при этом портал не нарушается.)

Все порталы неявно закрываются в конце транзакции, поэтому значение refcursor можно использовать для ссылки на открытый курсор только до конца транзакции.

40.7.3.1. FETCH

```
FETCH [направление { FROM | IN }] курсор INTO цель;
```

FETCH извлекает следующую строку из курсора в *цель*. В качестве *цели* может быть переменная-кортеж, переменная типа record или разделённый запятыми список простых переменных, как и в SELECT INTO. Если следующей строки нет, *цели* присваивается NULL. Как и в SELECT INTO, проверить, была ли получена запись, можно при помощи специальной переменной FOUND.

Здесь *направление* может быть любым допустимым в SQL-команде FETCH вариантом, кроме тех, что извлекают более одной строки. А именно: NEXT, PRIOR, FIRST, LAST, ABSOLUTE *число*, RELATIVE *число*, FORWARD или BACKWARD. Без указания *направления* подразумевается вариант NEXT. Везде, где используется *число*, оно может определяться любым целочисленным выражением (в отличие от SQL-команды FETCH, допускающей только целочисленные константы). Значения *направления*, которые требуют перемещения назад, приведут к ошибке, если курсор не был объявлен или открыт с указанием SCROLL.

курсор это переменная с типом refcursor, которая ссылается на открытый портал курсора.

Примеры:

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

40.7.3.2. MOVE

```
MOVE [направление { FROM | IN }] курсор;
```

MOVE перемещает курсор без извлечения данных. MOVE работает точно так же как и FETCH, но при этом только перемещает курсор и не извлекает строку, к которой переместился. Как и в SELECT INTO, проверить успешность перемещения можно с помощью специальной переменной FOUND.

Примеры:

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

40.7.3.3. UPDATE/DELETE WHERE CURRENT OF

```
UPDATE таблица SET ... WHERE CURRENT OF курсор;
DELETE FROM таблица WHERE CURRENT OF курсор;
```

Когда курсор позиционирован на строку таблицы, эту строку можно изменить или удалить при помощи курсора. Есть ограничения на то, каким может быть запрос курсора (в частности, не должно быть группировок), и крайне желательно использовать указание `FOR UPDATE`. За дополнительными сведениями обратитесь к странице справки DECLARE.

Пример:

```
UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

40.7.3.4. CLOSE

```
CLOSE курсор;
```

`CLOSE` закрывает связанный с курсором портал. Используется для того, чтобы освободить ресурсы раньше, чем закончится транзакция, или чтобы освободить курсорную переменную для повторного открытия.

Пример:

```
CLOSE curs1;
```

40.7.3.5. Возврат курсора из функции

Курсоры можно возвращать из функции на PL/pgSQL. Это полезно, когда нужно вернуть множество строк и столбцов, особенно если выборки очень большие. Для этого, в функции открывается курсор и его имя возвращается вызывающему (или просто открывается курсор, используя указанное имя портала, каким-либо образом известное вызывающему). Вызывающий затем может извлекать строки из курсора. Курсор может быть закрыт вызывающим или он будет автоматически закрыт при завершении транзакции.

Имя портала, используемое для курсора, может быть указано разработчиком или будет генерироваться автоматически. Чтобы указать имя портала, нужно просто присвоить строку в переменную `refcursor` перед его открытием. Значение строки переменной `refcursor` будет использоваться командой `OPEN` как имя портала. Однако если переменная `refcursor` имеет значение NULL, `OPEN` автоматически генерирует имя, которое не конфликтует с любым существующим порталом и присваивает его переменной `refcursor`.

> **Примечание**
>
> Связанная курсорная переменная инициализируется в строковое значение, представляющее собой имя самой переменной. Таким образом, имя портала совпадает с именем курсорной переменной, кроме случаев, когда разработчик переопределил имя, присвоив новое значение перед открытием курсора. Несвязанная курсорная переменная инициализируется в NULL и получит автоматически сгенерированное уникальное имя, если не будет переопределена.

Следующий пример показывает один из способов передачи имени курсора вызывающему:

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');
```

```
CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

В следующем примере используется автоматическая генерация имени курсора:

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
' LANGUAGE plpgsql;

-- для использования курсоров, необходимо начать транзакцию
BEGIN;
SELECT reffunc2();

      reffunc2
--------------------
 <unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

В следующем примере показан один из способов вернуть несколько курсоров из одной функции:

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;

-- для использования курсоров необходимо начать транзакцию
BEGIN;

SELECT * FROM myfunc('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

40.7.4. Обработка курсора в цикле

Один из вариантов цикла `FOR` позволяет перебирать строки, возвращённые курсором. Вот его синтаксис:

```
[ <<метка>> ]
FOR переменная-запись IN связанная_переменная_курсора [ ( [ имя_аргумента
 := ] значение_аргумента [, ...] ) ] LOOP
    операторы
END LOOP [ метка ];
```

Курсорная переменная должна быть связана с запросом при объявлении. Курсор *не может* быть открытым. Команда `FOR` автоматически открывает курсор и автоматически закрывает при завершении цикла. Список фактических значений аргументов должен присутствовать только в том случае, если курсор объявлялся с параметрами. Эти значения будут подставлены в запрос, как и при выполнении `OPEN` (см. Подраздел 40.7.2.3).

Данная *переменная-запись* автоматически определяется как переменная типа `record` и существует только внутри цикла (другие объявленные переменные с таким именем игнорируется в цикле). Каждая возвращаемая курсором строка последовательно присваивается этой переменной и выполняется тело цикла.

40.8. Сообщения и ошибки

40.8.1. Вывод сообщений и ошибок

Команда `RAISE` предназначена для вывода сообщений и вызова ошибок.

```
RAISE [ уровень ] 'формат' [, выражение [, ... ]] [ USING параметр = значение
 [, ... ] ];
RAISE [ уровень ] имя_условия [ USING параметр = выражение [, ... ] ];
RAISE [ уровень ] SQLSTATE 'sqlstate' [ USING параметр = выражение [, ... ] ];
RAISE [ уровень ] USING параметр = выражение [, ... ];
RAISE ;
```

уровень задаёт уровень важности ошибки. Возможные значения: `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING` и `EXCEPTION`. По умолчанию используется `EXCEPTION`. `EXCEPTION` вызывает ошибку (что обычно прерывает текущую транзакцию), остальные значения *уровня* только генерируют сообщения с различными уровнями приоритета. Будут ли сообщения конкретного приоритета переданы клиенту или записаны в журнал сервера, или и то, и другое, зависит от конфигурационных переменных log_min_messages и client_min_messages. За дополнительными сведениями обратитесь к Главе 18.

После указания *уровня*, если оно есть, можно задать строку *формата* (это должна быть простая строковая константа, не выражение). Строка формата определяет вид текста об ошибке, который будет выдан. За строкой формата могут следовать необязательные выражения аргументов, которые будут вставлены в сообщение. Внутри строки формата знак `%` заменяется строковым представлением значения очередного аргумента. Чтобы выдать символ `%` буквально, продублируйте его (как `%%`). Число аргументов должно совпадать с числом местозаполнителей `%` в строке формата, иначе при компиляции функции возникнет ошибка.

В следующем примере символ `%` будет заменён на значение `v_job_id`:

```
RAISE NOTICE 'Вызов функции cs_create_job(%)', v_job_id;
```

При помощи `USING` и последующих элементов *параметр* = *выражение* можно добавить дополнительную информацию к отчёту об ошибке. Все *выражения* представляют собой строковые выражения. Возможные ключевые слова для *параметра* следующие:

`MESSAGE`

Устанавливает текст сообщения об ошибке. Этот параметр не может использоваться, если в команде `RAISE` присутствует *формат* перед `USING`.

`DETAIL`

Предоставляет детальное сообщение об ошибке.

HINT

Предоставляет подсказку по вызванной ошибке.

ERRCODE

Устанавливает код ошибки (SQLSTATE). Код ошибки задаётся либо по имени, как показано в Приложении А, или напрямую, пятисимвольный код SQLSTATE.

COLUMN
CONSTRAINT
DATATYPE
TABLE
SCHEMA

Предоставляет имя соответствующего объекта, связанного с ошибкой.

Этот пример прерывает транзакцию и устанавливает сообщение об ошибке с подсказкой:

```
RAISE EXCEPTION 'Несуществующий ID --> %', user_id
      USING HINT = 'Проверьте ваш пользовательский ID';
```

Следующие два примера демонстрируют эквивалентные способы задания SQLSTATE:

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

У команды RAISE есть и другой синтаксис, в котором в качестве главного аргумента используется имя или код SQLSTATE ошибки. Например:

```
RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

Предложение USING в этом синтаксисе можно использовать для того, чтобы переопределить стандартное сообщение об ошибке, детальное сообщение, подсказку. Ещё один вариант предыдущего примера:

```
RAISE unique_violation USING MESSAGE = 'ID пользователя уже существует: ' || user_id;
```

Ещё один вариант — использовать RAISE USING или RAISE *уровень* USING, а всё остальное записать в списке USING.

И заключительный вариант, в котором RAISE не имеет параметров вообще. Эта форма может использоваться только в секции EXCEPTION блока и предназначена для того, чтобы повторно вызвать ошибку, которая сейчас перехвачена и обрабатывается.

Примечание

До версии PostgreSQL 9.1 команда RAISE без параметров всегда вызывала ошибку с выходом из блока, содержащего активную секцию EXCEPTION. Эту ошибку нельзя было перехватить, даже если RAISE в секции EXCEPTION поместить во вложенный блок со своей секцией EXCEPTION. Это было сочтено удивительным и не совместимым с Oracle PL/SQL.

Если в команде RAISE EXCEPTION не задано ни имя условия, ни код SQLSTATE, по умолчанию выдаётся исключение ERRCODE_RAISE_EXCEPTION (P0001). Если не задан текст сообщения, по умолчанию в качестве этого текста передаётся имя условия или код SQLSTATE.

Примечание

При задании SQLSTATE кода необязательно использовать только список предопределённых кодов ошибок. В качестве кода ошибки может быть любое пятисимвольное значение,

состоящее из цифр и/или ASCII символов в верхнем регистре, кроме 00000. Не рекомендуется использовать коды ошибок, которые заканчиваются на 000, потому что так обозначаются коды категорий. И чтобы их перехватить, нужно перехватывать целую категорию.

40.8.2. Проверка утверждений

Оператор ASSERT представляет удобное средство вставлять отладочные проверки в функции PL/pgSQL.

```
ASSERT условие [ , сообщение ];
```

Здесь *условие* — это логическое выражение, которое, как ожидается, должно быть всегда истинным; если это так, оператор ASSERT больше ничего не делает. Если же оно возвращает ложь или NULL, этот оператор выдаёт исключение ASSERT_FAILURE. (Если ошибка происходит при вычислении *условия*, она выдаётся как обычная ошибка.)

Если в нём задаётся необязательное *сообщение*, результат этого выражения (если он не NULL) заменяет сообщение об ошибке по умолчанию «assertion failed» (нарушение истинности), в случае, если *условие* не выполняется. В обычном случае, когда условие утверждения выполняется, выражение *сообщения* не вычисляется.

Проверку утверждений можно включить или отключить с помощью конфигурационного параметра plpgsql.check_asserts, принимающего логическое значение; по умолчанию она включена (on). Если этот параметр отключён (off), операторы ASSERT ничего не делают.

Учтите, что оператор ASSERT предназначен для выявления программных дефектов, а не для вывода обычных ошибок (для этого используется оператор RAISE, описанный выше).

40.9. Триггерные процедуры

В PL/pgSQL можно создавать триггерные процедуры, которые будут вызываться при изменениях данных или событиях в базе данных. Триггерная процедура создаётся командой CREATE FUNCTION, при этом у функции не должно быть аргументов, а типом возвращаемого значения должен быть trigger (для триггеров, срабатывающих при изменениях данных) или event_trigger (для триггеров, срабатывающих при событиях в базе). Для триггеров автоматически определяются специальные локальные переменные с именами вида TG_*имя*, описывающие условие, повлёкшее вызов триггера.

40.9.1. Триггеры при изменении данных

Триггер при изменении данных объявляется как функция без аргументов и с типом результата trigger. Заметьте, что эта функция должна объявляться без аргументов, даже если ожидается, что она будет получать аргументы, заданные в команде CREATE TRIGGER — такие аргументы передаются через TG_ARGV, как описано ниже.

Когда функция на PL/pgSQL срабатывает как триггер, в блоке верхнего уровня автоматически создаются несколько специальных переменных:

NEW

Тип данных RECORD. Переменная содержит новую строку базы данных для команд INSERT/UPDATE в триггерах уровня строки. В триггерах уровня оператора и для команды DELETE этой переменной значение не присваивается.

OLD

Тип данных RECORD. Переменная содержит старую строку базы данных для команд UPDATE/DELETE в триггерах уровня строки. В триггерах уровня оператора и для команды INSERT этой переменной значение не присваивается.

TG_NAME

Тип данных name. Переменная содержит имя сработавшего триггера.

TG_WHEN

Тип данных text. Строка, содержащая BEFORE, AFTER или INSTEAD OF, в зависимости от определения триггера.

TG_LEVEL

Тип данных text. Строка, содержащая ROW или STATEMENT, в зависимости от определения триггера.

TG_OP

Тип данных text. Строка, содержащая INSERT, UPDATE, DELETE или TRUNCATE, в зависимости от того, для какой операции сработал триггер.

TG_RELID

Тип данных oid. OID таблицы, для которой сработал триггер.

TG_RELNAME

Тип данных name. Имя таблицы, для которой сработал триггер. Эта переменная устарела и может стать недоступной в будущих релизах. Вместо неё нужно использовать TG_TABLE_NAME.

TG_TABLE_NAME

Тип данных name. Имя таблицы, для которой сработал триггер.

TG_TABLE_SCHEMA

Тип данных name. Имя схемы, содержащей таблицу, для которой сработал триггер.

TG_NARGS

Тип данных integer. Число аргументов в команде CREATE TRIGGER, которые передаются в триггерную процедуру.

TG_ARGV[]

Тип данных массив text. Аргументы от оператора CREATE TRIGGER. Индекс массива начинается с 0. Для недопустимых значений индекса (< 0 или >= tg_nargs) возвращается NULL.

Триггерная функция должна вернуть либо NULL, либо запись/строку, соответствующую структуре таблице, для которой сработал триггер.

Если BEFORE триггер уровня строки возвращает NULL, то все дальнейшие действия с этой строкой прекращаются (т. е. не срабатывают последующие триггеры, команда INSERT/UPDATE/DELETE для этой строки не выполняется). Если возвращается не NULL, то дальнейшая обработка продолжается именно с этой строкой. Возвращение строки отличной от начальной NEW, изменяет строку, которая будет вставлена или изменена. Поэтому, если в триггерной функции нужно выполнить некоторые действия и не менять саму строку, то нужно возвратить переменную NEW (или её эквивалент). Для того чтобы изменить сохраняемую строку, можно поменять отдельные значения в переменной NEW и затем её вернуть. Либо создать и вернуть полностью новую переменную. В случае строчного триггера BEFORE для команды DELETE само возвращаемое значение не имеет прямого эффекта, но оно должно быть отличным от NULL, чтобы не прерывать обработку строки. Обратите внимание, что переменная NEW всегда NULL в триггерах на DELETE, поэтому возвращать её не имеет смысла. Традиционной идиомой для триггеров DELETE является возврат переменной OLD.

Триггеры INSTEAD OF (это всегда триггеры уровня строк и они могут применяться только с представлениями) могут возвращать NULL, чтобы показать, что они не выполняли никаких изменений, так что обработку этой строки можно не продолжать (то есть, не вызывать последующие триггеры и не считать строку в числе обработанных строк для окружающих команд INSERT/UPDATE/DELETE). В противном случае должно быть возвращено значение, отличное от NULL, показывающее, что триггер выполнил запрошенную операцию. Для операций INSERT и UPDATE возвращаемым значением должно быть NEW, которое триггерная функция может модифицировать для поддержки предложений INSERT RETURNING и UPDATE RETURNING (это также повлияет на значение строки, передаваемое последующим триггерам, или доступное под специальным псевдонимом EXCLUDED в операторе INSERT с предложением ON CONFLICT DO UPDATE). Для операций DELETE возвращаемым значением должно быть OLD.

Возвращаемое значение для строчного триггера AFTER и триггеров уровня оператора (BEFORE или AFTER) всегда игнорируется. Это может быть и NULL. Однако в этих триггерах по-прежнему можно прервать вызвавшую их команду, для этого нужно явно вызвать ошибку.

Пример 40.3 показывает пример триггерной процедуры в PL/pgSQL.

Пример 40.3. Триггерная процедура PL/pgSQL

Триггер, показанный в этом примере, при любом добавлении или изменении строки в таблице сохраняет в этой строке информацию о текущем пользователе и отметку времени. Кроме того, он требует, чтобы было указано имя сотрудника и зарплата задавалась положительным числом.

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Проверить, что указаны имя сотрудника и зарплата
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;

        -- Кто будет работать, если за это надо будет платить?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;

        -- Запомнить, кто и когда изменил запись
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Другой вариант вести журнал изменений для таблицы предполагает создание новой таблицы, которая будет содержать отдельную запись для каждой выполненной команды INSERT, UPDATE,

DELETE. Этот подход можно рассматривать как протоколирование изменений таблицы для аудита. Пример 40.4 показывает реализацию соответствующей триггерной процедуры в PL/pgSQL.

Пример 40.4. Триггерная процедура для аудита в PL/pgSQL

Показанный в этом примере триггер гарантирует, что любое добавление, изменение или удаление строки в таблице `emp` будет зафиксировано в таблице `emp_audit` (для аудита). Также он фиксирует текущее время, имя пользователя и тип выполняемой операции.

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- Добавление строки в emp_audit, которая отражает операцию, выполняемую в emp;
        -- для определения типа операции применяется специальная переменная TG_OP.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
        END IF;
        RETURN NULL; -- возвращаемое значение для триггера AFTER игнорируется
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```

У предыдущего примера есть разновидность, которая использует представление, соединяющее основную таблицу и таблицу аудита, для отображения даты последнего изменения каждой строки. При этом подходе по-прежнему ведётся полный журнал аудита в отдельной таблице, но также имеется представление с упрощенным аудиторским следом. Это представление содержит временную метку, которая вычисляется для каждой строки из данных аудиторской таблицы. Пример 40.5 показывает пример триггера на представление для аудита в PL/pgSQL.

Пример 40.5. Триггер на PL/pgSQL для аудита в представлении

В этом примере триггер, связанный с представлением, делает это представление изменяемым и гарантирует, что любая команда на добавление, изменение или удаление строки в представлении будет записана для аудита в таблицу `emp_audit`. Также записываются временная метка, имя пользователя и тип выполняемой операции. Представление показывает дату последнего изменения для каждой строки.

```
CREATE TABLE emp (
    empname           text PRIMARY KEY,
    salary            integer
```

```
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary            integer,
    stamp             timestamp NOT NULL
);

CREATE VIEW emp_view AS
    SELECT e.empname,
           e.salary,
           max(ea.stamp) AS last_updated
      FROM emp e
      LEFT JOIN emp_audit ea ON ea.empname = e.empname
     GROUP BY 1, 2;

CREATE OR REPLACE FUNCTION update_emp_view() RETURNS TRIGGER AS $$
    BEGIN
        --
        -- Выполнить требуемую операцию в emp и добавить в emp_audit строку,
        -- отражающую эту операцию.
        --
        IF (TG_OP = 'DELETE') THEN
            DELETE FROM emp WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            OLD.last_updated = now();
            INSERT INTO emp_audit VALUES('D', user, OLD.*);
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            UPDATE emp SET salary = NEW.salary WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('U', user, NEW.*);
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp VALUES(NEW.empname, NEW.salary);

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('I', user, NEW.*);
            RETURN NEW;
        END IF;
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_view
    FOR EACH ROW EXECUTE PROCEDURE update_emp_view();
```

Один из вариантов использования триггеров это поддержание в актуальном состоянии отдельной таблицы итогов для некоторой таблицы. В некоторых случаях отдельная таблица с итогами может использоваться в запросах вместо основной таблицы. При этом зачастую время выполнения запросов значительно сокращается. Эта техника широко используется в хранилищах данных, где таблицы фактов могут быть очень большими. Пример 40.6 показывает триггерную процедуру в PL/pgSQL, которая поддерживает таблицу итогов для таблицы фактов в хранилище данных.

Пример 40.6. Триггерная процедура на PL/pgSQL для ведения таблицы итогов

Представленная здесь схема данных частично основана на примере *Grocery Store* из книги *The Data Warehouse Toolkit* Ральфа Кимбалла (Ralph Kimball).

```
--
-- Основные таблицы: таблица временных периодов и таблица фактов продаж
--
CREATE TABLE time_dimension (
    time_key                    integer NOT NULL,
    day_of_week                 integer NOT NULL,
    day_of_month                integer NOT NULL,
    month                       integer NOT NULL,
    quarter                     integer NOT NULL,
    year                        integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
    time_key                    integer NOT NULL,
    product_key                 integer NOT NULL,
    store_key                   integer NOT NULL,
    amount_sold                 numeric(12,2) NOT NULL,
    units_sold                  integer NOT NULL,
    amount_cost                 numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- Таблица с итогами продаж по периодам
--
CREATE TABLE sales_summary_bytime (
    time_key                    integer NOT NULL,
    amount_sold                 numeric(15,2) NOT NULL,
    units_sold                  numeric(12) NOT NULL,
    amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--
-- Функция и триггер для пересчёта столбцов итогов при выполнении
-- команд INSERT, UPDATE, DELETE
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER
AS $maint_sales_summary_bytime$
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
        delta_amount_cost       numeric(15,2);
    BEGIN

        -- Вычислить изменение количества/суммы.
        IF (TG_OP = 'DELETE') THEN

            delta_time_key = OLD.time_key;
            delta_amount_sold = -1 * OLD.amount_sold;
            delta_units_sold = -1 * OLD.units_sold;
            delta_amount_cost = -1 * OLD.amount_cost;
```

```
ELSIF (TG_OP = 'UPDATE') THEN

    -- Запретить изменение time_key -
    -- (это ограничение не должно вызвать неудобств, так как
    -- в основном изменения будут выполняться по схеме DELETE + INSERT).
    IF ( OLD.time_key != NEW.time_key) THEN
        RAISE EXCEPTION 'Запрещено изменение time_key : % -> %',
                                              OLD.time_key, NEW.time_key;
    END IF;

    delta_time_key = OLD.time_key;
    delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
    delta_units_sold = NEW.units_sold - OLD.units_sold;
    delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

ELSIF (TG_OP = 'INSERT') THEN

    delta_time_key = NEW.time_key;
    delta_amount_sold = NEW.amount_sold;
    delta_units_sold = NEW.units_sold;
    delta_amount_cost = NEW.amount_cost;

END IF;


-- Внести новые значения в существующую строку итогов или
-- добавить новую.
<<insert_update>>
LOOP
    UPDATE sales_summary_bytime
        SET amount_sold = amount_sold + delta_amount_sold,
            units_sold = units_sold + delta_units_sold,
            amount_cost = amount_cost + delta_amount_cost
        WHERE time_key = delta_time_key;

    EXIT insert_update WHEN found;

    BEGIN
        INSERT INTO sales_summary_bytime (
                    time_key,
                    amount_sold,
                    units_sold,
                    amount_cost)
            VALUES (
                    delta_time_key,
                    delta_amount_sold,
                    delta_units_sold,
                    delta_amount_cost
                   );

        EXIT insert_update;

    EXCEPTION
        WHEN UNIQUE_VIOLATION THEN
            -- ничего не делать
    END;
END LOOP insert_update;
```

```
        RETURN NULL;

    END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
    FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;
```

Триггеры `AFTER` также могут использовать *переходные таблицы* для просмотра всего набора строк, изменённых оператором, вызвавшим триггер. Команда `CREATE TRIGGER` назначает имена одной или обеим переходным таблицам, а затем функция может по этим именам обращаться к ним как к временным таблицам только для чтения. Это иллюстрирует Пример 40.7.

Пример 40.7. Организация аудита с переходными таблицами

В данном примере достигается тот же результат, что и в Пример 40.4, но вместо триггера, срабатывающего для каждой строки, в нём используется триггер, срабатывающий единожды для оператора и получающий нужные ему данные в переходной таблице. Это может быть гораздо быстрее, чем вариант с построчным триггером, когда целевой оператор изменяет сразу множество строк. Заметьте, что мы должны объявить отдельные триггеры для каждого вида события, так как предложения `REFERENCING` в каждом случае будут разными. Но это не мешает при желании использовать одну триггерную функцию. (На практике может быть лучше использовать три отдельные функции и не проверять `TG_OP` во время выполнения.)

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- Добавление строк в emp_audit, которые отражают операции, выполняемые в emp;
        -- для определения типа операций применяется специальная переменная TG_OP.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit
                SELECT 'D', now(), user, o.* FROM old_table o;
```

```
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit
                SELECT 'U', now(), user, n.* FROM new_table n;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit
                SELECT 'I', now(), user, n.* FROM new_table n;
        END IF;
        RETURN NULL; -- возвращаемое значение для триггера AFTER игнорируется
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit_ins
    AFTER INSERT ON emp
    REFERENCING NEW TABLE AS new_table
    FOR EACH STATEMENT EXECUTE PROCEDURE process_emp_audit();
CREATE TRIGGER emp_audit_upd
    AFTER UPDATE ON emp
    REFERENCING OLD TABLE AS old_table NEW TABLE AS new_table
    FOR EACH STATEMENT EXECUTE PROCEDURE process_emp_audit();
CREATE TRIGGER emp_audit_del
    AFTER DELETE ON emp
    REFERENCING OLD TABLE AS old_table
    FOR EACH STATEMENT EXECUTE PROCEDURE process_emp_audit();
```

40.9.2. Триггеры событий

В PL/pgSQL можно создавать событийные триггеры. Postgres Pro требует, чтобы процедура, которая вызывается как событийный триггер, объявлялась без аргументов и типом возвращаемого значения был `event_trigger`.

Когда функция на PL/pgSQL вызывается как событийный триггер, в блоке верхнего уровня автоматически создаются несколько специальных переменных:

`TG_EVENT`

Тип данных `text`. Строка, содержащая событие, для которого сработал триггер.

`TG_TAG`

Тип данных `text`. Переменная, содержащая тег команды, для которой сработал триггер.

Пример 40.8 показывает пример процедуры событийного триггера в PL/pgSQL.

Пример 40.8. Процедура событийного триггера в PL/pgSQL

Триггер в этом примере просто выдаёт сообщение `NOTICE` каждый раз, когда выполняется поддерживаемая команда.

```
CREATE OR REPLACE FUNCTION snitch() RETURNS event_trigger AS $$
BEGIN
    RAISE NOTICE 'Произошло событие: % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER snitch ON ddl_command_start EXECUTE PROCEDURE snitch();
```

40.10. PL/pgSQL изнутри

В этом разделе обсуждаются некоторые детали реализации, которые пользователям PL/pgSQL важно знать.

40.10.1. Подстановка переменных

SQL-операторы и выражения внутри функции на PL/pgSQL могут ссылаться на переменные и параметры этой функции. За кулисами PL/pgSQL заменяет параметры запросов для таких ссылок. Параметры будут заменены только в местах, где параметр или ссылка на столбец синтаксически допустимы. Как крайний случай, рассмотрим следующий пример плохого стиля программирования:

```
INSERT INTO foo (foo) VALUES (foo);
```

Первый раз `foo` появляется на том месте, где синтаксически должно быть имя таблицы, поэтому замены не будет, даже если функция имеет переменную `foo`. Второй раз `foo` встречается там, где должно быть имя столбца таблицы, поэтому замены не будет и здесь. Только третье вхождение `foo` является кандидатом на то, чтобы быть ссылкой на переменную функции.

> **Примечание**
>
> Версии PostgreSQL до 9.0 пытаются заменить переменную во всех трёх случаях, что приводит к синтаксической ошибке.

Если имена переменных синтаксически не отличаются от названий столбцов таблицы, то возможна двусмысленность и в ссылках на таблицы. Является ли данное имя ссылкой на столбец таблицы или ссылкой на переменную? Изменим предыдущий пример:

```
INSERT INTO dest (col) SELECT foo + bar FROM src;
```

Здесь `dest` и `src` должны быть именами таблиц, `col` должен быть столбцом `dest`. Однако `foo` и `bar` могут быть как переменными функции, так и столбцами `src`.

По умолчанию, PL/pgSQL выдаст ошибку, если имя в операторе SQL может относиться как к переменной, так и к столбцу таблицы. Ситуацию можно исправить переименованием переменной, переименованием столбца, точной квалификацией неоднозначной ссылки или указанием PL/pgSQL машине, какую интерпретацию предпочесть.

Самое простое решение — переименовать переменную или столбец. Общее правило кодирования предполагает использование различных соглашений о наименовании для переменных PL/pgSQL и столбцов таблиц. Например, если имена переменных всегда имеют вид `v_`*`имя`*, а имена столбцов никогда не начинаются на `v_`, то конфликты исключены.

В качестве альтернативы можно дополнить имена неоднозначных ссылок, чтобы сделать их точными. В приведённом выше примере `src.foo` однозначно бы определялась, как ссылка на столбец таблицы. Чтобы сделать однозначной ссылку на переменную, переменная должна быть объявлена в блоке с меткой, и далее нужно использовать эту метку (см. Раздел 40.2). Например:

```
<<block>>
DECLARE
    foo int;
BEGIN
    foo := ...;
    INSERT INTO dest (col) SELECT block.foo + bar FROM src;
```

Здесь `block.foo` ссылается на переменную, даже если в таблице `src` есть столбец `foo`. Параметры функции, а также специальные переменные, такие как `FOUND`, могут быть дополнены именем функции, потому что они неявно объявлены во внешнем блоке, метка которого совпадает с именем функции.

Иногда может быть не очень практичным исправлять таким способом все неоднозначные ссылки в большом куске PL/pgSQL кода. В таких случаях можно указать, чтобы PL/pgSQL разрешал неоднозначные ссылки в пользу переменных (это совместимо с PL/pgSQL до версии PostgreSQL 9.0), или в пользу столбцов таблицы (совместимо с некоторыми другими системами, такими как Oracle).

На уровне всей системы поведение PL/pgSQL регулируется установкой конфигурационного параметра `plpgsql.variable_conflict`, имеющего значения: `error`, `use_variable` или `use_column` (`error` устанавливается по умолчанию при установке системы). Изменение этого параметра влияет на все последующие компиляции операторов в функциях на PL/pgSQL, но не на операторы уже скомпилированные в текущей сессии. Так как изменение этого параметра может привести к неожиданным изменениям в поведении функций на PL/pgSQL, он может быть изменён только суперпользователем.

Поведение PL/pgSQL можно изменять для каждой отдельной функции, если добавить в начало функции одну из этих специальных команд:

```
#variable_conflict error
#variable_conflict use_variable
#variable_conflict use_column
```

Эти команды влияют только на функцию, в которой они записаны и перекрывают действие `plpgsql.variable_conflict`. Пример:

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    #variable_conflict use_variable
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = curtime, comment = comment
          WHERE users.id = id;
    END;
$$ LANGUAGE plpgsql;
```

В команде `UPDATE`, `curtime`, `comment` и `id` будут ссылаться на переменные и параметры функции вне зависимости от того, есть ли столбцы с такими именами в таблице `users`. Обратите внимание, что нужно дополнить именем таблицы ссылку на `users.id` в предложении `WHERE`, чтобы она ссылалась на столбец таблицы. При этом необязательно дополнять ссылку на `comment` в левой части списка `UPDATE`, так как синтаксически в этом месте должно быть имя столбца таблицы `users`. Эту функцию можно было бы записать и без зависимости от значения `variable_conflict`:

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    <<fn>>
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = fn.curtime, comment = stamp_user.comment
          WHERE users.id = stamp_user.id;
    END;
$$ LANGUAGE plpgsql;
```

Замена переменных не происходит в строке, исполняемой командой `EXECUTE` или её вариантом. Если нужно вставлять изменяющиеся значения в такую команду, то это делается либо при построении самой командной строки или с использованием `USING`, как показано в Подразделе 40.5.4.

Замена переменных в настоящее время работает только в командах `SELECT`, `INSERT`, `UPDATE` и `DELETE`, потому что основная SQL машина допускает использование параметров запроса только в этих командах. Чтобы использовать изменяемые имена или значения в других типах операторов (обычно называемых служебными), необходимо составить текст команды в виде строки и выполнить её в `EXECUTE`.

40.10.2. Кеширование плана

Интерпретатор PL/pgSQL анализирует исходный текст функции и строит внутреннее бинарное дерево инструкций при первом вызове функции (для каждой сессии). В дерево инструкций полностью переводится вся структура операторов PL/pgSQL, но для выражений и команд SQL, используемых в функции, это происходит не сразу.

При первом выполнении в функции каждого выражения или команды SQL интерпретатор PL/pgSQL разбирает и анализирует команду для создания подготовленного к выполнению оператора с помощью функции `SPI_prepare` менеджера интерфейса программирования сервера. Последующие обращения к этому выражению или команде повторно используют подготовленный к выполнению оператор. Таким образом, SQL-команды, находящиеся в редко посещаемой ветке кода условного оператора, не несут накладных расходов на разбор команд, если они так и не будут выполнены в текущей сессии. Здесь есть недостаток, заключающийся в том, что ошибки в определённом выражении или команде не могут быть обнаружены, пока выполнение не дойдёт до этой части функции. (Тривиальные синтаксические ошибки обнаружатся в ходе первоначального разбора, но ничего более серьёзного не будет обнаружено до исполнения.)

Кроме того, PL/pgSQL (точнее, менеджер интерфейса программирования сервера) будет пытаться кешировать план выполнения для любого подготовленного к исполнению оператора. При каждом вызове оператора, если не используется план из кеша, генерируется новый план выполнения, и текущие значения параметров (то есть значения переменных PL/pgSQL) могут быть использованы для оптимизации нового плана. Если оператор не имеет параметров или выполняется много раз, менеджер интерфейса программирования сервера рассмотрит вопрос о создании и кешировании (для повторного использования) общего плана, не зависящего от значений параметров. Как правило, это происходит в тех случаях, когда план выполнения не очень чувствителен к имеющимся ссылкам на значения переменных PL/pgSQL. В противном случае выгоднее каждый раз формировать новый план. Более подробно поведение подготовленных операторов рассматривается в PREPARE.

Чтобы PL/pgSQL мог сохранять подготовленные операторы и планы выполнения, команды SQL в коде PL/pgSQL, должны использовать одни и те же таблицы и столбцы при каждом исполнении. А это значит, что в SQL-командах нельзя использовать названия таблиц и столбцов в качестве параметров. Чтобы обойти это ограничение, нужно сконструировать динамическую команду для оператора PL/pgSQL `EXECUTE` — ценой будет разбор и построение нового плана выполнения при каждом вызове.

Изменчивая природа переменных типа `record` представляет ещё одну проблему в этой связи. Когда поля переменной типа `record` используются в выражениях или операторах, типы данных полей не должны меняться от одного вызова функции к другому, так как при анализе каждого выражения будет использоваться тот тип данных, который присутствовал при первом вызове. При необходимости можно использовать `EXECUTE` для решения этой проблемы.

Если функция используется в качестве триггера более чем для одной таблицы, PL/pgSQL независимо подготавливает и кеширует операторы для каждой такой таблицы. То есть создаётся кеш для каждой комбинации триггерная функция + таблица, а не только для каждой функции. Это устраняет некоторые проблемы, связанные с различными типами данных. Например, триггерная функция сможет успешно работать со столбцом `key`, даже если в разных таблицах этот столбец имеет разные типы данных.

Таким же образом, функции с полиморфными типами аргументов имеют отдельный кеш для каждой комбинации фактических типов аргументов, так что различия типов данных не вызывают неожиданных сбоев.

Кеширование операторов иногда приводит к неожиданным эффектам при интерпретации чувствительных ко времени значений. Например, есть разница между тем, что делают эти две функции:

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS void AS $$
    BEGIN
        INSERT INTO logtable VALUES (logtxt, 'now');
    END;
$$ LANGUAGE plpgsql;
```

и

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS void AS $$
    DECLARE
```

```
        curtime timestamp;
    BEGIN
        curtime := 'now';
        INSERT INTO logtable VALUES (logtxt, curtime);
    END;
$$ LANGUAGE plpgsql;
```

В случае `logfunc1`, при анализе `INSERT`, основной анализатор Postgres Pro знает, что строку `'now'` следует толковать как `timestamp`, потому что целевой столбец таблицы `logtable` имеет такой тип данных. Таким образом, `'now'` будет преобразовано в константу `timestamp` при анализе `INSERT`, а затем эта константа будет использоваться в последующих вызовах `logfunc1` в течение всей сессии. Разумеется, это не то, что хотел программист. Лучше было бы использовать функцию `now()` или `current_timestamp`.

В случае `logfunc2`, основной анализатор Postgres Pro не знает, какого типа будет `'now'` и поэтому возвращает значение типа `text`, содержащее строку `now`. При последующем присваивании локальной переменной `curtime` интерпретатор PL/pgSQL приводит эту строку к типу `timestamp`, вызывая функции `textout` и `timestamp_in`. Таким образом, метка времени будет обновляться при каждом выполнении, как и ожидается программистом. И хотя всё работает как ожидалось, это ужасно неэффективно, поэтому использование функции `now()` по-прежнему значительно лучше.

40.11. Советы по разработке на PL/pgSQL

Хороший способ разрабатывать на PL/pgSQL заключается в том, чтобы в одном окне с текстовым редактором по выбору создавать тексты функций, а в другом окне с psql загружать и тестировать эти функции. В таком случае удобно записывать функцию, используя `CREATE OR REPLACE FUNCTION`. Таким образом, можно легко загрузить файл для обновления определения функции. Например:

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
          ....
$$ LANGUAGE plpgsql;
```

В psql, можно загрузить или перезагрузить такой файл определения функции, выполнив:

```
\i filename.sql
```

а затем сразу выполнять команды SQL для тестирования функции.

Ещё один хороший способ разрабатывать на PL/pgSQL связан с использованием GUI инструментов, облегчающих разработку на процедурном языке. Один из примеров такого инструмента pgAdmin, хотя есть и другие. Такие инструменты часто предоставляют удобные возможности, такие как экранирование одинарных кавычек, отладка и повторное создание функций.

40.11.1. Обработка кавычек

Код функции на PL/pgSQL указывается в команде `CREATE FUNCTION` в виде строки. Если записывать строку как обычно, внутри одинарных кавычек, то любой символ одинарной кавычки должен дублироваться, так же как и должен дублироваться каждый знак обратной косой черты (если используется синтаксис с экранированием в строках). Дублирование кавычек в лучшем случае утомительно, а в более сложных случаях код может стать совершенно непонятным, так как легко может потребоваться четыре или более идущих подряд кавычек. Вместо этого при создании тела функции рекомендуется использовать знаки доллара в качестве кавычек (см. Подраздел 4.1.2.4). При таком подходе никогда не потребуется дублировать кавычки, но придётся позаботиться о том, чтобы иметь разные долларовые разделители для каждого уровня вложенности. Например, команду `CREATE FUNCTION` можно записать так:

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
          ....
$PROC$ LANGUAGE plpgsql;
```

Внутри можно использовать кавычки для простых текстовых строк и `$$` для разграничения фрагментов SQL-команды, собираемой из отдельных строк. Если нужно взять в кавычки текст, который включает `$$`, можно использовать `$Q$`, и так далее.

Следующая таблица показывает, как применяются знаки кавычек, если не используется экранирование долларами. Это может быть полезно при переводе кода, не использующего экранирование знаками доллара, в нечто более понятное.

1 кавычка

В начале и конце тела функции, например:

```
CREATE FUNCTION foo() RETURNS integer AS '
            ....
' LANGUAGE plpgsql;
```

Внутри такой функции любая кавычка *должна* дублироваться.

2 кавычки

Для строковых литералов внутри тела функции, например:

```
a_output := ''Blah'';
SELECT * FROM users WHERE f_name=''foobar'';
```

При использовании знаков доллара можно просто написать:

```
a_output := 'Blah';
SELECT * FROM users WHERE f_name='foobar';
```

и именно это увидит исполнитель PL/pgSQL в обоих случаях.

4 кавычки

Когда нужны одинарные кавычки в строковой константе внутри тела функции, например:

```
a_output := a_output || '' AND name LIKE ''''foobar'''' AND xyz''
```

К `a_output` будет добавлено: `AND name LIKE 'foobar' AND xyz`

При использовании знаков доллара это записывается так:

```
a_output := a_output || $$ AND name LIKE 'foobar' AND xyz$$
```

будьте внимательны, при этом не должно быть внешнего долларового разделителя `$$`.

6 кавычек

Когда нужны одинарные кавычки в строковой константе внутри тела функции, при этом кавычки находятся в конце строковой константы. Например:

```
a_output := a_output || '' AND name LIKE ''''foobar''''''
```

К `a_output` будет добавлено: `AND name LIKE 'foobar'`.

При использовании знаков доллара это записывается так:

```
a_output := a_output || $$ AND name LIKE 'foobar'$$
```

10 кавычек

Когда нужны две одиночные кавычки в строковой константе (это уже 8 кавычек), примыкающие к концу строковой константы (ещё 2). Вероятно, такое может понадобиться при разработке функции, которая генерирует другие функции, как показано в Примере 40.10. Например:

```
a_output := a_output || '' if v_'' ||
    referrer_keys.kind || '' like ''''''''''
    || referrer_keys.key_string || ''''''''''
    then return ''''''  || referrer_keys.referrer_type
    || ''''''; end if;'';
```

Значение `a_output` затем будет:

```
if v_... like ''...'' then return ''...''; end if;
```

При использовании знаков доллара:

```
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
    || referrer_keys.key_string || $$'
    then return '$$  || referrer_keys.referrer_type
    || $$'; end if;$$;
```

где предполагается, что нужны только одиночные кавычки в `a_output`, так как потребуется повторное взятие в кавычки перед использованием.

40.11.2. Дополнительные проверки во время компиляции

Чтобы помочь найти и предупредить простые, но часто встречающиеся проблемы, PL/PgSQL предоставляет дополнительные *проверки*. Если они включены в конфигурации, то во время компиляции функций будут выдаваться дополнительные сообщения `WARNING` или ошибки `ERROR`. Функция, при компиляции которой выдавалось `WARNING`, при последующем выполнении не будет выдавать это сообщение и её можно протестировать в отдельной среде разработки.

Для включения этих проверок используются параметры конфигурации `plpgsql.extra_warnings` для предупреждений и `plpgsql.extra_errors` для ошибок. Каждому из параметров можно присвоить список значений, разделённых запятыми, значение `"none"` или `"all"`. По умолчанию используется `"none"`. В настоящий момент доступна только одна проверка:

`shadowed_variables`

Проверяет, что объявление новой переменной не скрывает ранее объявленную переменную.

Следующий пример показывает эффект присваивания `plpgsql.extra_warnings` значения `shadowed_variables`:

```
SET plpgsql.extra_warnings TO 'shadowed_variables';

CREATE FUNCTION foo(f1 int) RETURNS int AS $$
DECLARE
f1 int;
BEGIN
RETURN f1;
END;
$$ LANGUAGE plpgsql;
WARNING:  variable "f1" shadows a previously defined variable
LINE 3: f1 int;
        ^
CREATE FUNCTION
```

40.12. Портирование из Oracle PL/SQL

В этом разделе рассматриваются различия между языками Postgres Pro PL/pgSQL и Oracle PL/SQL, чтобы помочь разработчикам, переносящим приложения из Oracle® в Postgres Pro.

PL/pgSQL во многих аспектах похож на PL/SQL . Это блочно-структурированный, императивный язык, в котором все переменные должны объявляться. Присваивания, циклы и условные операторы в обоих языках похожи. Основные отличия, которые необходимо иметь в виду при портировании с PL/SQL в PL/pgSQL, следующие:

- Если имя, используемое в SQL-команде, может быть как именем столбца таблицы, так и ссылкой на переменную функции, то PL/SQL считает, что это имя столбца таблицы. Это соответствует поведению PL/pgSQL при `plpgsql.variable_conflict = use_column`, что не является значением по умолчанию, как описано в Подразделе 40.10.1. В первую очередь, было бы правильно избегать таких двусмысленностей, но если требуется портировать большое количество кода, зависящее от данного поведения, то установка переменной `variable_conflict` может быть лучшим решением.
- В Postgres Pro тело функции должно быть записано в виде строки. Поэтому нужно использовать знак доллара в качестве кавычек или экранировать одиночные кавычки в теле функции. (См. Подраздел 40.11.1.)

- Имена типов данных часто требуют корректировки. Например, в Oracle строковые значения часто объявляются с типом `varchar2`, не являющимся стандартным типом SQL. В Postgres Pro вместо него нужно использовать `varchar` или `text`. Подобным образом, тип `number` нужно заменять на `numeric` или другой числовой тип, если найдётся более подходящий.
- Для группировки функций вместо пакетов используются схемы.
- Так как пакетов нет, нет и пакетных переменных. Это несколько раздражает. Вместо этого можно хранить состояние каждого сеанса во временных таблицах.
- Целочисленные циклы `FOR` с указанием `REVERSE` работают по-разному. В PL/SQL значение счётчика уменьшается от второго числа к первому, в то время как в PL/pgSQL счётчик уменьшается от первого ко второму. Поэтому при портировании нужно менять местами границы цикла. Это печально, но вряд ли будет изменено. (См. Подраздел 40.6.3.5.)
- Циклы `FOR` по запросам (не курсорам) также работают по-разному. Переменная цикла должна быть объявлена, в то время как в PL/SQL она объявляется неявно. Преимущество в том, что значения переменных доступны и после выхода из цикла.
- Существуют некоторые отличия в нотации при использовании курсорных переменных.

40.12.1. Примеры портирования

Пример 40.9 показывает, как портировать простую функцию из PL/SQL в PL/pgSQL.

Пример 40.9. Портирование простой функции из PL/SQL в PL/pgSQL

Функция Oracle PL/SQL:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar2,
                                                  v_version varchar2)
RETURN varchar2 IS
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
/
show errors;
```

Пройдемся по этой функции и посмотрим различия по сравнению с PL/pgSQL:

- Имя типа `varchar2` нужно сменить на `varchar` или `text`. В примерах данного раздела мы будем использовать `varchar`, но обычно лучше выбрать `text`, если не требуется ограничивать длину строк.
- Ключевое слово `RETURN` в прототипе функции (не в теле функции) заменяется на `RETURNS` в Postgres Pro. Кроме того, `IS` становится `AS`, и нужно добавить предложение `LANGUAGE`, потому что PL/pgSQL — не единственный возможный язык.
- В Postgres Pro тело функции является строкой, поэтому нужно использовать кавычки или знаки доллара. Это заменяет завершающий `/` в подходе Oracle.
- Команда `show errors` не существует в Postgres Pro и не требуется, так как ошибки будут выводиться автоматически.

Вот как эта функция будет выглядеть после портирования в Postgres Pro:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                                  v_version varchar)
RETURNS varchar AS $$
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
```

```
    RETURN v_name || '/' || v_version;
END;
$$ LANGUAGE plpgsql;
```

Пример 40.10 показывает, как портировать функцию, которая создаёт другую функцию, и как обрабатывать проблемы с кавычками.

Пример 40.10. Портирование функции, создающей другую функцию, из PL/SQL в PL/pgSQL

Следующая процедура получает строки из `SELECT` и строит большую функцию, в целях эффективности возвращающую результат в операторах `IF`.

Версия Oracle:

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_cmd VARCHAR(4000);
BEGIN
    func_cmd := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN VARCHAR2,
                 v_domain IN VARCHAR2, v_url IN VARCHAR2) RETURN VARCHAR2 IS BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_cmd := func_cmd ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ''' || referrer_key.key_string
          || ''' THEN RETURN ''' || referrer_key.referrer_type
          || '''; END IF;';
    END LOOP;

    func_cmd := func_cmd || ' RETURN NULL; END;';

    EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;
```

В конечном итоге в Postgres Pro эта функция может выглядеть так:

```
CREATE OR REPLACE FUNCTION cs_update_referrer_type_proc() RETURNS void AS $func$
DECLARE
    referrer_keys CURSOR IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_body text;
    func_cmd text;
BEGIN
    func_body := 'BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_body := func_body ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ' || quote_literal(referrer_key.key_string)
          || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
          || '; END IF;' ;
    END LOOP;

    func_body := func_body || ' RETURN NULL; END;';

    func_cmd :=
```

```
      'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
                                                        v_domain varchar,
                                                        v_url varchar)
        RETURNS varchar AS '
      || quote_literal(func_body)
      || ' LANGUAGE plpgsql;' ;

    EXECUTE func_cmd;
END;
$func$ LANGUAGE plpgsql;
```

Обратите внимание, что тело функции строится отдельно, с использованием `quote_literal` для дублирования кавычек. Эта техника необходима, потому что мы не можем безопасно использовать знаки доллара при определении новой функции: мы не знаем наверняка, какие строки будут вставлены из `referrer_key.key_string`. (Мы предполагаем, что `referrer_key.kind` всегда имеет значение из списка: `host`, `domain` или `url`, но `referrer_key.key_string` может быть чем угодно, в частности, может содержать знаки доллара.) На самом деле, в этой функций есть улучшение по сравнению с оригиналом Oracle, потому что не будет генерироваться неправильный код, когда `referrer_key.key_string` или `referrer_key.referrer_type` содержат кавычки.

Пример 40.11 показывает, как портировать функцию с выходными параметрами (`OUT`) и манипулирующую строками. В Postgres Pro нет встроенной функции `instr`, но её можно создать, используя комбинацию других функций. В Подраздел 40.12.3 приведена реализации `instr` на PL/pgSQL, которая может быть полезна вам при портировании ваших функций.

Пример 40.11. Портирование из PL/SQL в PL/pgSQL процедуры, которая манипулирует строками и содержит `OUT` параметры

Следующая процедура на языке Oracle PL/SQL разбирает URL и возвращает составляющие его элементы (сервер, путь и запрос).

Версия Oracle:

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
    v_url IN VARCHAR2,
    v_host OUT VARCHAR2,  -- Возвращается как результат
    v_path OUT VARCHAR2,  -- И это тоже
    v_query OUT VARCHAR2) -- И это
IS
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);
```

```
    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

Вот возможная трансляция в PL/pgSQL:

```
CREATE OR REPLACE FUNCTION cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- Возвращается как результат
    v_path OUT VARCHAR,  -- И это тоже
    v_query OUT VARCHAR) -- И это
AS $$
DECLARE
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
$$ LANGUAGE plpgsql;
```

Эту функцию можно использовать так:

```
SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');
```

Пример 40.12 показывает, как портировать процедуру, использующую большое количество специфических для Oracle возможностей.

Пример 40.12. Портирование процедуры из PL/SQL в PL/pgSQL

Версия Oracle:

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
    PRAGMA AUTONOMOUS_TRANSACTION; -- 1
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE; -- 2

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- снять блокировку 3
        raise_application_error(-20000,
                'Не удалось создать новое задание. Задание сейчас выполняется.');
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN dup_val_on_index THEN NULL; -- ничего не делать, если задание уже есть
    END;
    COMMIT;
END;
/
show errors
```

Подобные процедуры легко преобразуются в функции Postgres Pro, возвращающие `void`. На примере этой процедуры можно научиться следующему:

1. В Postgres Pro нет оператора `PRAGMA`.
2. Если выполнить `LOCK TABLE` в PL/pgSQL, блокировка не будет снята, пока не завершится вызывающая транзакция.
3. В функции на PL/pgSQL нельзя использовать `COMMIT`. Функция работает в рамках некоторой внешней транзакции, и поэтому `COMMIT` будет означать прекращение выполнения функции. Однако в данном конкретном случае в этом нет необходимости, потому что блокировка, полученная командой `LOCK TABLE`, будет снята при вызове ошибки.

В PL/pgSQL эту процедуру можно портировать так:

```
CREATE OR REPLACE FUNCTION cs_create_job(v_job_id integer) RETURNS void AS $$
DECLARE
    a_running_job_count integer;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        RAISE EXCEPTION 'Не удалось создать новое задание. Задание сейчас
 выполняется.'; -- 1
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
```

```
        WHEN unique_violation THEN -- 2
            -- ничего не делать, если задание уже есть
    END;
END;
$$ LANGUAGE plpgsql;
```

1 Синтаксис `RAISE` существенно отличается от Oracle, хотя основной вариант `RAISE` *`имя_исключения`* работает похоже.

2 Имена исключений, поддерживаемые PL/pgSQL, отличаются от исключений в Oracle. Количество встроенных имён исключений значительно больше (см. Приложение А). В настоящее время нет способа задать пользовательское имя исключения, хотя вместо этого можно вызывать ошибку с заданным пользователем значением SQLSTATE.

Основное функциональное отличие между этой процедурой и Oracle эквивалента в том, что монопольная блокировка таблицы `cs_jobs` будет продолжаться до окончания вызывающей транзакции. Кроме того, если впоследствии работа вызывающей программы прервётся (например из-за ошибки), произойдёт откат всех действий, выполненных в этой процедуре.

40.12.2. На что ещё обратить внимание

В этом разделе рассматриваются ещё несколько вещей, на которые нужно обращать внимание при портировании функций из Oracle PL/SQL в Postgres Pro.

40.12.2.1. Неявный откат изменений после возникновения исключения

В PL/pgSQL при перехвате исключения в секции `EXCEPTION` все изменения в базе данных с начала блока автоматически откатываются. В Oracle это эквивалентно следующему:

```
BEGIN
    SAVEPOINT s1;
    ... здесь код ...
EXCEPTION
    WHEN ... THEN
        ROLLBACK TO s1;
        ... здесь код ...
    WHEN ... THEN
        ROLLBACK TO s1;
        ... здесь код ...
END;
```

При портировании процедуры Oracle, которая использует `SAVEPOINT` и `ROLLBACK TO` в таком же стиле, задача простая: достаточно убрать операторы `SAVEPOINT` и `ROLLBACK TO`. Если же `SAVEPOINT` и `ROLLBACK TO` используются по-другому, то придётся подумать.

40.12.2.2. `EXECUTE`

PL/pgSQL версия `EXECUTE` работает аналогично версии в PL/SQL, но нужно помнить об использовании `quote_literal` и `quote_ident`, как описано в Подразделе 40.5.4. Без использования этих функций конструкции типа `EXECUTE 'SELECT * FROM $1';` будут работать ненадёжно.

40.12.2.3. Оптимизация функций на PL/pgSQL

Для оптимизации исполнения Postgres Pro предоставляет два модификатора при создании функции: «изменчивость» (будет ли функция всегда возвращать тот же результат при тех же аргументах) и «строгость» (возвращает ли функция NULL, если хотя бы один из аргументов NULL). Для получения подробной информации обратитесь к справочной странице CREATE FUNCTION.

При использовании этих атрибутов оптимизации оператор `CREATE FUNCTION` может выглядеть примерно так:

```
CREATE FUNCTION foo(...) RETURNS integer AS $$
...
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

40.12.3. Приложение

Этот раздел содержит код для совместимых с Oracle функций `instr`, которые можно использовать для упрощения портирования.

```
--
-- instr functions that mimic Oracle's counterpart
-- Syntax: instr(string1, string2 [, n [, m]])
-- where [] denotes optional parameters.
--
-- Search string1, beginning at the nth character, for the mth occurrence
-- of string2.  If n is negative, search backwards, starting at the abs(n)'th
-- character from the end of string1.
-- If n is not passed, assume 1 (search starts at first character).
-- If m is not passed, assume 1 (find first occurrence).
-- Returns starting index of string2 in string1, or 0 if string2 is not found.
--

CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS $$
BEGIN
    RETURN instr($1, $2, 1);
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                      beg_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        temp_str := substring(string FROM beg_index);
        pos := position(string_to_search_for IN temp_str);

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN pos + beg_index - 1;
        END IF;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search_for);
        length := char_length(string);
        beg := length + 1 + beg_index;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            IF string_to_search_for = temp_str THEN
                RETURN beg;
            END IF;

            beg := beg - 1;
        END LOOP;
```

```
        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                      beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
    ss_length integer;
BEGIN
    IF occur_index <= 0 THEN
        RAISE 'argument ''%'' is out of range', occur_index
          USING ERRCODE = '22003';
    END IF;

    IF beg_index > 0 THEN
        beg := beg_index - 1;
        FOR i IN 1..occur_index LOOP
            temp_str := substring(string FROM beg + 1);
            pos := position(string_to_search_for IN temp_str);
            IF pos = 0 THEN
                RETURN 0;
            END IF;
            beg := beg + pos;
        END LOOP;

        RETURN beg;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search_for);
        length := char_length(string);
        beg := length + 1 + beg_index;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            IF string_to_search_for = temp_str THEN
                occur_number := occur_number + 1;
                IF occur_number = occur_index THEN
                    RETURN beg;
                END IF;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
```

```
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

Глава 41. PL/Tcl — процедурный язык Tcl

PL/Tcl — это загружаемый процедурный язык для СУБД Postgres Pro, позволяющий использовать *язык Tcl* для написания функций и триггерных процедур.

41.1. Обзор

PL/Tcl предоставляет большинство возможностей, которые имеет разработчик функций на C, с небольшими ограничениями, и позволяет применять мощные библиотеки обработки строк, существующие для Tcl.

Одним убедительным *хорошим* ограничением является то, что весь код выполняется в контексте безопасности интерпретатора Tcl. Помимо ограниченного набора команд безопасного Tcl, разрешены только несколько команд для обращения к базе данных через SPI и вызовы `elog()` для выдачи сообщений. PL/Tcl не даёт возможности взаимодействовать с внутренним механизмом сервера баз данных или обращаться к ОС с правами серверного процесса Postgres Pro, что возможно в функциях на C. Таким образом, использование этого языка можно доверить непривилегированным пользователям; это не даст им неограниченные полномочия.

Ещё одно существенное ограничение заключается в том, что функции на Tcl нельзя использовать для создания функций ввода/вывода для новых типов данных.

Иногда возникает желание написать функцию на Tcl, которая не будут ограничена безопасным Tcl. Например, может потребоваться функция, которая будет посылать сообщения по почте. Для этих случаев есть вариация PL/Tcl, названная `PL/TclU` (название подразумевает «untrusted Tcl», недоверенный Tcl). Это тот же язык, за исключением того, что для него используется полноценный интерпретатор Tcl. *Если применяется PL/TclU, он должен быть установлен как недоверенный процедурный язык*, чтобы только суперпользователи могли создавать функции на нём. Автор функции на PL/TclU должен позаботиться о том, чтобы эту функцию нельзя было использовать не по назначению, так как она может делать всё, что может пользователь с правами администратора баз данных.

Разделяемый объектный код для обработчиков вызова PL/Tcl и PL/TclU собирается автоматически и устанавливается в каталог библиотек Postgres Pro, если поддержка Tcl включена на этапе конфигурирования процедуры установки. Чтобы установить PL/Tcl и/или PL/TclU в конкретную базу данных, воспользуйтесь командой `CREATE EXTENSION`, например, так: `CREATE EXTENSION pltcl` или `CREATE EXTENSION pltclu`.

41.2. Функции на PL/Tcl и их аргументы

Чтобы создать функцию на языке PL/Tcl, используйте стандартный синтаксис CREATE FUNCTION:

```
CREATE FUNCTION имя_функции (типы_аргументов) RETURNS тип_результата AS $$
    # Тело функции на PL/Tcl
$$ LANGUAGE pltcl;
```

С PL/TclU команда та же, но в качестве языка должно быть указано `pltclu`.

Тело функции содержит просто скрипт на Tcl. Когда вызывается функция, значения аргументов передаются скрипту Tcl в виде переменных с именами `1` ... *n*. Результат из кода Tcl возвращается как обычно, оператором `return`.

Например, функцию, возвращающую большее из двух целых чисел, можно определить так:

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl STRICT;
```

Обратите внимание на предложение `STRICT`, которое избавляет нас от необходимости думать о входящих значениях NULL: если при вызове передаётся значение NULL, функция не будет выполняться вовсе, будет сразу возвращён результат NULL.

В нестрогой функции, если фактическое значение аргумента — NULL, соответствующей переменной `$n` будет присвоена пустая строка. Чтобы определить, был ли передан NULL в определённом аргументе, используйте функцию `argisnull`. Например, предположим, что нам нужна функция `tcl_max`, которая с одним аргументом NULL и вторым аргументом не NULL должна возвращать не NULL, а второй аргумент:

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {[argisnull 1]} {
        if {[argisnull 2]} { return_null }
        return $2
    }
    if {[argisnull 2]} { return $1 }
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl;
```

Как показано выше, чтобы вернуть значение NULL из функции PL/Tcl, нужно выполнить `return_null`. Это можно сделать и в строгой, и в нестрогой функции.

Аргументы составного типа передаются функции в виде массивов Tcl. Именами элементов массива являются имена атрибутов составного типа. Если атрибут в переданной строке имеет значение NULL, он будет отсутствовать в данном массиве. Например:

```
CREATE TABLE employee (
    name text,
    salary integer,
    age integer
);

CREATE FUNCTION overpaid(employee) RETURNS boolean AS $$
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
        return "t"
    }
    return "f"
$$ LANGUAGE pltcl;
```

Функции PL/Tcl могут возвращать и результаты составного типа. Для этого код на Tcl должен вернуть список пар имя/значение столбца, соответствующий ожидаемому типу результата. Столбцы, имена которых в этом списке отсутствуют, получат значения NULL, а если в списке указано имя несуществующего столбца, возникнет ошибка. Например:

```
CREATE FUNCTION square_cube(in int, out squared int, out cubed int) AS $$
    return [list squared [expr {$1 * $1}] cubed [expr {$1 * $1 * $1}]]
$$ LANGUAGE pltcl;
```

Подсказка

Список результатов можно создать из желаемого кортежа, представленного в виде массива, с помощью команды `array get` языка Tcl. Например:

```
CREATE FUNCTION raise_pay(employee, delta int) RETURNS employee AS $$
    set 1(salary) [expr {$1(salary) + $2}]
    return [array get 1]
```

```
$$ LANGUAGE pltcl;
```

Функции PL/Tcl могут возвращать наборы результатов. Для этого код на Tcl должен вызывать `return_next` для каждой возвращаемой строки, передавая ей соответствующее значение, когда возвращается скалярный тип, или список пар имя/значение столбца, когда возвращается составной тип. Пример с результатом скалярного типа:

```
CREATE FUNCTION sequence(int, int) RETURNS SETOF int AS $$
    for {set i $1} {$i < $2} {incr i} {
        return_next $i
    }
$$ LANGUAGE pltcl;
```

и с результатом составного:

```
CREATE FUNCTION table_of_squares(int, int) RETURNS TABLE (x int, x2 int) AS $$
    for {set i $1} {$i < $2} {incr i} {
        return_next [list x $i x2 [expr {$i * $i}]]
    }
$$ LANGUAGE pltcl;
```

41.3. Значения данных в PL/Tcl

Значения аргументов, передаваемые в код функции PL/Tcl, представляют собой просто входные аргументы, преобразованные в текстовый вид (так же, как при выводе оператором `SELECT`). И наоборот, команды `return` и `return_next` примут любую строку, соответствующую формату ввода для объявленного типа результата функции или заданного столбца в результате составного типа.

41.4. Глобальные данные в PL/Tcl

Иногда полезно иметь некоторые глобальные данные, сохраняемые между двумя вызовами функции или совместно используемые разными функциями. Это легко сделать в PL/Tcl, но есть некоторые ограничения, которые необходимо понимать.

По соображениям безопасности, PL/Tcl выполняет функции, вызываемые некоторой ролью SQL в отдельном интерпретаторе Tcl, выделенном для этой роли. Это предотвращает случайное или злонамеренное влияние одного пользователя на поведение функций PL/Tcl другого пользователя. В каждом интерпретаторе будут свои значения всех «глобальных» переменных Tcl. Таким образом, в двух функциях PL/Tcl будут общие глобальные переменные, только если они выполняются одной ролью SQL. В приложении, выполняющем код в одном сеансе с разными ролями SQL (вызывающем функции `SECURITY DEFINER`, использующем команду `SET ROLE` и т. д.) может понадобиться явно предпринять дополнительные меры, чтобы функции могли разделять свои данные. Для этого сначала установите для функций, которые должны взаимодействовать, одного владельца, а затем задайте для них свойство `SECURITY DEFINER`. Разумеется, при этом нужно позаботиться о том, чтобы эти функции не могли сделать ничего непредусмотренного.

Все функции PL/TclU, вызываемые в одном сеансе, выполняются одним интерпретатором Tcl, который, конечно, отличается от интерпретатора(ов), используемого для функций PL/Tcl. Поэтому глобальные данные функций PL/TclU автоматически становятся общими. Это не считается угрозой безопасности, так как все функции PL/TclU выполняются на одном уровне доверия, а именно уровне суперпользователя базы данных.

Чтобы защитить функции PL/Tcl от непреднамеренного влияния друг на друга, каждой из них предоставляется глобальная переменная-массив через команду `upvar`. Глобальным именем этой переменной является внутреннее имя функции, а в качестве локального выбрано `GD`. Переменную `GD` рекомендуется использовать для постоянных внутренних данных функции. Обычные глобальные переменные Tcl следует использовать только для значений, которые предназначены именно для совместного использования несколькими функциями. (Заметьте, что

массивы GD являются глобальными только для конкретного интерпретатора, так что они не нарушают ограничения безопасности, описанные выше.)

Использование GD демонстрируется в примере spi_execp, приведённом ниже.

41.5. Обращение к базе данных из PL/Tcl

В этом разделе для обозначения необязательных элементов в описании синтаксиса используются знаки вопроса (а не квадратные скобки), как это принято в Tcl. Имеются следующие команды для доступа к базе данных из тела функции PL/Tcl:

spi_exec ?-count *n*? ?-array *имя*? *команда* ?*тело-цикла*?

Выполняет команду SQL, заданную в виде строки. В случае ошибки в этой команде выдаётся ошибка в Tcl. В противном случае spi_exec возвращает число обработанных командой строк (выбранных, добавленных, изменённых или удалённых), либо ноль, если эта команда — служебный оператор. Кроме того, если команда — оператор SELECT, значения выбранных столбцов помещаются в переменные Tcl, как описано ниже.

Необязательное значение -count задаёт для spi_exec максимальное число строк, которое должно быть обработано в команде. Его действие можно представить как выполнение FETCH *n* для курсора, предварительно подготовленного для команды.

Если в качестве команды выполняется оператор SELECT, значения результирующих столбцов помещаются в переменные Tcl, названные по именам столбцов. Если передаётся -array, значения столбцов вместо этого становятся элементами названного ассоциативного массива, индексами в котором становятся имена столбцов. Кроме того, в элементе с именем «.tupno» сохраняется номер текущей строки в результирующем наборе (отсчитывая от нуля), если только это имя не занято одним из столбцов результата.

Если в качестве команды выполняется SELECT без указания скрипта *тело-цикла*, в переменных Tcl или элементах массива сохраняется только первая строка результатов; оставшиеся строки (если они есть), игнорируются. Если запрос не возвращает строки, не сохраняется ничего. (Этот случай можно отследить, проверив результат spi_exec.) Например, команда:

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

присвоит переменной $cnt в Tcl число строк, содержащихся в системном каталоге pg_proc.

Если передаётся необязательный аргумент *тело-цикла*, заданный в нём блок скрипта Tcl будет выполняться для каждой строки результата запроса. (Аргумент *тело-цикла* игнорируется, если целевая команда — не SELECT.) При этом значения столбцов текущей строки сохраняются в переменных Tcl или элементах массива перед каждой итерацией этого цикла. Например, код:

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "have table $C(relname)"
}
```

будет выводить в журнал сообщение для каждой строки pg_class. Это работает подобно другим конструкциям циклов в Tcl; в частности, команды continue и break в теле цикла будут действовать обычным образом.

Если в столбце результата запроса выдаётся NULL, целевая переменная для неё не устанавливается, и оказывается «неустановленной».

spi_prepare *запрос список-типов*

Подготавливает и сохраняет план запроса для последующего выполнения. Сохранённый план будет продолжать существование до завершения текущего сеанса.

Запрос может принимать параметры, то есть местозаполнители для значений, которые будут передаваться, когда план будет собственно выполняться. В строке запроса эти параметры

обозначаются как `$1` ... `$n`. Если в запросе используются параметры, нужно задать имена типов этих параметров в виде списка Tcl. (Если параметры отсутствуют, задайте пустой *список_типов*.)

Функция `spi_prepare` возвращает идентификатор запроса, который может использоваться в последующих вызовах `spi_execp`. Пример приведён в описании `spi_execp`.

`spi_execp ?-count` *n*`? ?-array` *имя*`? ?-nulls` *строка*`?` *ид-запроса* `?`*список-значений*`? ?`*тело-цикла*`?`

Выполняет запрос, ранее подготовленный функцией `spi_prepare`. В качестве *ид_запроса* передаётся идентификатор, возвращённый функцией `spi_prepare`. Если в запросе задействуются параметры, необходимо указать *список-значений*. Это должен быть принятый в Tcl список параметров. Он должен иметь ту же длину, что и список типов параметров, ранее переданный `spi_prepare`. Опустите *список-значений*, если у запроса нет параметров.

Необязательный аргумент `-nulls` принимает строку из пробелов и символов `'n'`, которые отмечают, в каких параметрах `spi_execp` передаются значения NULL. Если присутствует, эта строка должна иметь ту же длину, что и *список-значений*. В случае её отсутствия значения всех параметров считаются отличными от NULL.

Не считая отличий в способе передачи запроса и параметров, `spi_execp` работает так же, как `spi_exec`. Параметры `-count`, `-array` и *тело-цикла* задаются так же, и так же передаётся возвращаемое значение.

Взгляните на пример функции на PL/Tcl, использующей подготовленный план:

```
CREATE FUNCTION t1_count(integer, integer) RETURNS integer AS $$
    if {![ info exists GD(plan) ]} {
        # подготовить сохранённый план при первом вызове
        set GD(plan) [ spi_prepare \
                "SELECT count(*) AS cnt FROM t1 WHERE num >= \$1 AND num <= \$2" \
                [ list int4 int4 ] ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
$$ LANGUAGE pltcl;
```

Обратные косые черты внутри строки запроса, передаваемой функции `spi_prepare`, нужны для того, чтобы маркеры `$n` передавались функции `spi_prepare` как есть, а не заменялись при подстановке переменных Tcl.

`spi_lastoid`

Возвращает OID строки, вставленной последней командой `spi_exec` или `spi_execp`, если этой командой был оператор `INSERT` с одной строкой и изменяемая таблица содержит OID. (В противном случае вы получите ноль.)

`subtransaction` *команда*

Скрипт Tcl, который содержит *команда*, выполняется в подтранзакции SQL. Если этот скрипт возвращает ошибку, вся подтранзакция откатывается назад, а затем в окружающий код Tcl возвращается ошибка. За дополнительными подробностями и примером обратитесь к Разделу 41.9.

`quote` *строка*

Дублирует все вхождения апострофа и обратной косой черты в заданной строке. Это можно использовать для защиты строк, которые будут вставляться в команды SQL, передаваемые в `spi_exec` или `spi_prepare`. Например, представьте, что при выполнении такой команды SQL:

```
"SELECT '$val' AS ret"
```

переменная языка Tcl `val` содержит `doesn't`. Это приведёт к формированию такой окончательной строки команды:

```
SELECT 'doesn't' AS ret
```

при разборе которой в процессе `spi_exec` или `spi_prepare` возникнет ошибка. Чтобы этот запрос работал правильно, итоговая команда должна выглядеть так:

```
SELECT 'doesn''t' AS ret
```

Получить её в PL/Tcl можно так:

```
"SELECT '[ quote $val ]' AS ret"
```

Преимуществом `spi_execp` является то, что для неё заключать значения параметров в кавычки подобным образом не нужно, так как параметры никогда не разбираются в составе строки команды SQL.

`elog` *уровень сообщение*

Выдаёт служебное сообщение или сообщение об ошибке. Возможные уровни сообщений: `DEBUG` (ОТЛАДКА), `LOG` (СООБЩЕНИЕ), `INFO` (ИНФОРМАЦИЯ), `NOTICE` (ЗАМЕЧАНИЕ), `WARNING` (ПРЕДУПРЕЖДЕНИЕ), `ERROR` (ОШИБКА) и `FATAL` (ВАЖНО). С уровнем `ERROR` выдаётся ошибка; если она не перехватывается окружающим кодом Tcl, она распространяется в вызывающий запрос, что приводит к прерыванию текущей транзакции или подтранзакции. По сути то же самое делает команда `error` языка Tcl. Сообщение уровня `FATAL` прерывает транзакцию и приводит к завершению текущего сеанса. (Вероятно, нет обоснованной причины использовать этот уровень ошибок в функциях PL/Tcl, но он поддерживается для полноты.) При использовании других уровней происходит просто вывод сообщения с заданным уровнем важности. Будут ли сообщения определённого уровня передаваться клиенту и/или записываться в журнал, определяется конфигурационными переменными log_min_messages и client_min_messages. За дополнительными сведениями обратитесь к Главе 18 и Разделу 41.8.

41.6. Процедуры триггеров на PL/Tcl

На PL/Tcl можно написать триггерные процедуры. Postgres Pro требует, чтобы процедура, которая будет вызываться как триггерная, была объявлена как функция без аргументов и возвращала тип `trigger`.

Информация от менеджера триггеров передаётся в тело процедуры в следующих переменных:

`$TG_name`

Имя триггера из оператора `CREATE TRIGGER`.

`$TG_relid`

Идентификатор объекта таблицы, для которой будет вызываться триггерная процедура.

`$TG_table_name`

Имя таблицы, для которой будет вызываться триггерная процедура.

`$TG_table_schema`

Схема таблицы, для которой будет вызываться триггерная процедура.

`$TG_relatts`

Список языка Tcl, содержащий имена столбцов таблицы. В начало списка добавлен пустой элемент, поэтому при поиске в этом списке имени столбца с помощью стандартной в Tcl команды `lsearch` будет возвращён номер элемента, начиная с 1, так же, как нумеруются столбцы в Postgres Pro. (В позициях удалённых столбцов также содержатся пустые элементы, так что нумерация следующих за ними атрибутов не нарушается.)

`$TG_when`

Строка `BEFORE`, `AFTER` или `INSTEAD OF`, в зависимости от типа события триггера.

`$TG_level`

Строка `ROW` или `STATEMENT`, в зависимости от уровня события триггера.

`$TG_op`

Строка `INSERT`, `UPDATE`, `DELETE` или `TRUNCATE`, в зависимости от действия события триггера.

`$NEW`

Ассоциативный массив, содержащий значения новой строки таблицы для действий `INSERT` или `UPDATE`, либо пустой массив для `DELETE`. Индексами в массиве являются имена столбцов. Столбцы со значениями NULL в нём отсутствуют. Для триггеров уровня оператора этот массив не определяется.

`$OLD`

Ассоциативный массив, содержащий значения старой строки таблицы для действий `UPDATE` или `DELETE`, либо пустой массив для `INSERT`. Индексами в массиве являются имена столбцов. Столбцы со значениями NULL в нём отсутствуют. Для триггеров уровня оператора этот массив не определяется.

`$args`

Список на языке Tcl аргументов процедуры, заданных в операторе `CREATE TRIGGER`. Эти аргументы также доступны под обозначениями `$1` ... `$n` в теле процедуры.

Возвращаемым значением триггерной процедуры может быть строка `OK` или `SKIP` либо список пар имя столбца/значение. Если возвращается значение `OK`, операция (`INSERT`/`UPDATE`/`DELETE`), которая привела к срабатыванию триггера, выполняется нормально. Значение `SKIP` указывает менеджеру триггеров просто пропустить эту операцию с текущей строкой данных. Если возвращается список, через него PL/Tcl передаёт менеджеру триггеров изменённую строку; содержимое изменённой строки задаётся именами и значениями столбцов в списке. Все столбцы, не перечисленные в этом списке, получают значения NULL. Возвращать изменённую строку имеет смысл только для триггеров уровня строки с порядком `BEFORE` команд `INSERT` и `UPDATE`, в которых вместо заданной в `$NEW` будет записываться изменённая строка; либо с порядком `INSTEAD OF` команд `INSERT` и `UPDATE`, в которых возвращаемая строка служит исходными данными для предложений `INSERT RETURNING` или `UPDATE RETURNING`. В триггерах уровня строки с порядком `BEFORE` или `INSTEAD OF` команды `DELETE` возврат изменённой строки воспринимается так же, как и возврат значения `OK`, то есть операция выполняется. Для всех остальных типов триггеров возвращаемое значение игнорируется.

> **Подсказка**
>
> Список результатов можно создать из изменённого кортежа, представленного в виде массива, с помощью команды `array get` языка Tcl.

Следующий небольшой пример показывает триггерную процедуру, которая ведёт в таблице целочисленный счётчик числа изменений, выполненных в строке. Для новых строк счётчик инициализируется нулевым значением, а затем увеличивается на единицу при каждом изменении.

```
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS $$
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
```

```
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
        default {
            return OK
        }
    }
    return [array get NEW]
$$ LANGUAGE pltcl;

CREATE TABLE mytab (num integer, description text, modcnt integer);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
    FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

Заметьте, что сама триггерная процедура не знает имени столбца; оно передаётся в аргументах триггера. Это позволяет применять эту триггерную процедуру для различных таблиц.

41.7. Процедуры событийных триггеров в PL/Tcl

На PL/Tcl можно написать процедуры событийных триггеров. Postgres Pro требует, чтобы процедура, которая будет вызываться как событийный триггер, была объявлена как функция без аргументов и возвращала тип `event_trigger`.

Информация от менеджера триггеров передаётся в тело процедуры в следующих переменных:

`$TG_event`

Имя события, при котором срабатывает этот триггер.

`$TG_tag`

Тег команды, для которой срабатывает этот триггер.

Возвращаемое значение триггерной процедуры игнорируется.

В этом примере мини-процедура событийного триггера просто выдаёт замечание (`NOTICE`) при каждом выполнении поддерживаемой команды:

```
CREATE OR REPLACE FUNCTION tclsnitch() RETURNS event_trigger AS $$
  elog NOTICE "tclsnitch: $TG_event $TG_tag"
$$ LANGUAGE pltcl;

CREATE EVENT TRIGGER tcl_a_snitch ON ddl_command_start EXECUTE PROCEDURE tclsnitch();
```

41.8. Обработка ошибок в PL/Tcl

Tcl-код, содержащийся или вызываемый из функции PL/Tcl, может выдавать ошибку либо выполняя недопустимую операцию, либо генерируя ошибку с помощью команды `error` языка Tcl или команды `elog` языка PL/Tcl. Такие ошибки могут быть перехвачены в среде Tcl с помощью команды Tcl `catch`. Если ошибка не перехватывается, а распространяется выше уровня выполнения функций PL/Tcl, она передаётся в запрос, вызвавший функцию, как ошибка SQL.

И напротив, ошибки СУБД, возникающие внутри команд `spi_exec`, `spi_prepare` и `spi_execp` в среде PL/Tcl, выдаются как ошибки Tcl, так что их можно перехватить командой Tcl `catch`. (Каждая из этих команд PL/Tcl выполняет SQL-операцию в подтранзакции, которая откатывается в случае ошибки, так что для частично завершённых операций производится автоматическая очистка.) Опять же, если ошибка не перехватывается и распространяется выше верхнего уровня, она становится ошибкой SQL.

В Tcl имеется переменная `errorCode`, представляющая дополнительную информацию об ошибке в виде, удобном для обработки в программах на Tcl. Эта информация передаётся в формате

списка Tcl, первое слово в котором указывает на подсистему или библиотеку, выдающую ошибку; последующее содержимое определяется в зависимости от подсистемы или библиотеки. Для ошибок СУБД, возникающих в командах PL/Tcl, первым словом будет `POSTGRES`, вторым — номер версии Postgres Pro, а дополнительные слова представляют пары имя/значения, передающие подробную информацию об ошибке. В этих парах всегда передаются поля `SQLSTATE`, `condition` и `message` (первые два представляют код ошибки и имя условия, как описано в Приложении А). Также могут передаваться поля `detail`, `hint`, `context`, `schema`, `table`, `column`, `datatype`, `constraint`, `statement`, `cursor_position`, `filename`, `lineno` и `funcname`.

С информацией в переменной `errorCode` среды PL/Tcl удобно работать, загрузив переменную в массив, чтобы имена полей стали индексами в массиве. Пример такого кода:

```
if {[catch { spi_exec $sql_command }]} {
    if {[lindex $::errorCode 0] == "POSTGRES"} {
        array set errorArray $::errorCode
        if {$errorArray(condition) == "undefined_table"} {
            # разобраться с отсутствием таблицы
        } else {
            # разобраться с другими типами ошибок SQL
        }
    }
}
```

(Двойные двоеточия явно указывают, что переменная `errorCode` является глобальной.)

41.9. Явные подтранзакции в PL/Tcl

Перехват ошибок, произошедших при обращении к базе данных, как описано в Разделе 41.8, может привести к нежелательной ситуации, когда часть операций будет успешно выполнена, прежде чем произойдёт сбой. Данные останутся в несогласованном состоянии после обработки такой ошибки. PL/Tcl предлагает решение этой проблемы в форме явных подтранзакций.

Рассмотрите функцию, реализующую перевод денег между двумя счетами:

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
    if [catch {
        spi_exec "UPDATE accounts SET balance = balance - 100 WHERE account_name =
 'joe'"
        spi_exec "UPDATE accounts SET balance = balance + 100 WHERE account_name =
 'mary'"
    } errormsg] {
        set result [format "error transferring funds: %s" $errormsg]
    } else {
        set result "funds transferred successfully"
    }
    spi_exec "INSERT INTO operations (result) VALUES ('[quote $result]')"
$$ LANGUAGE pltcl;
```

Если второй оператор `UPDATE` выдаст исключение, эта функция запишет в журнал сообщение об ошибке, но результат первого `UPDATE` будет тем не менее зафиксирован. Другими словами, денежные средства будут списаны со счёта Джо, но не поступят на счёт Мери. Это происходит потому, что каждый вызов `spi_exec` выполняется в отдельной подтранзакции, а откатывается только одна из подтранзакций.

В таких случаях вы можете обернуть несколько операций с базой данных в одну явную подтранзакцию, которая будет выполнена успешно или отменена как единое целое. Для этого в PL/Tcl есть команда `subtransaction`. С ней мы можем переписать нашу функцию так:

```
CREATE FUNCTION transfer_funds2() RETURNS void AS $$
    if [catch {
        subtransaction {
```

```
            spi_exec "UPDATE accounts SET balance = balance - 100 WHERE account_name =
 'joe'"
            spi_exec "UPDATE accounts SET balance = balance + 100 WHERE account_name =
 'mary'"
        }
    } errormsg] {
        set result [format "error transferring funds: %s" $errormsg]
    } else {
        set result "funds transferred successfully"
    }
    spi_exec "INSERT INTO operations (result) VALUES ('[quote $result]')"
$$ LANGUAGE pltcl;
```

Заметьте, что и в этом случае нужно использовать `catch`. В противном случае ошибка распространится на верхний уровень функции, что не даст произвести желаемое добавление записи в таблицу `operations`. Команда `subtransaction` не перехватывает ошибки, она только обеспечивает откат всех операций с базой данных в своей области действия в случае ошибки.

Откат явной подтранзакции происходит в случае любых ошибок, сгенерированных вложенным кодом Tcl, а не только ошибок, возникающих при обращении к базе данных. Таким образом, обычное исключение Tcl, возникшее внутри команды `subtransaction`, также приведёт к откату подтранзакции. Однако при выходе из вложенного кода Tcl без ошибки (например, с помощью команды `return`) откат не производится.

41.10. Конфигурация PL/Tcl

В этом разделе описываются параметры конфигурации, влияющие на работу PL/Tcl.

`pltcl.start_proc` (`string`)

В этом параметре, если он не пуст, задаётся имя (возможно, дополненное схемой) функции на языке PL/Tcl без параметров, которая будет выполняться, когда для PL/Tcl будет создаваться новый экземпляр Tcl. Такая функция может выполнять инициализацию в рамках сеанса, например, загружать дополнительный код Tcl. Новый интерпретатор Tcl создаётся при первом выполнении какой-либо функции PL/Tcl в сеансе базы данных или когда требуется дополнительный интерпретатор из-за того, что функция PL/Tcl была вызвана новой ролью SQL.

Указанная функция должна быть написана на языке `pltcl` и не должна иметь свойство `SECURITY DEFINER`. (Благодаря этим ограничениям эта функция будет запускаться в интерпретаторе, который она должна инициализировать.) Текущий пользователь должен иметь право и на её выполнение тоже.

Если эта функция завершится ошибкой, эта ошибка прервёт вызов функции, которой потребовался новый интерпретатор, и распространится в вызывающий запрос, приводя к прерыванию текущей транзакции или подтранзакции. Любые действия, уже произведённые в среде Tcl, отменены не будут; однако этот интерпретатор более не будет использоваться. При следующей попытке использования этого языка последует повторная попытка инициализации со свежим интерпретатором Tcl.

Изменять этот параметр разрешено только суперпользователям. Хотя изменить его можно в рамках сеанса, такие изменения не повлияют на работу интерпретаторов Tcl, созданных ранее.

`pltclu.start_proc` (`string`)

Это параметр полностью аналогичен `pltcl.start_proc`, но применяется к PL/TclU. Указанная функция должна быть написана на языке `pltclu`.

41.11. Имена процедур Tcl

В Postgres Pro одно имя функции может использоваться разными определениями функций, если они имеют разное число и типы аргументов. Tcl, однако, требует, чтобы имена всех процедур

различались. PL/Tcl решает эту проблему, устанавливая такие внутренние имена процедур Tcl, чтобы они включали в свой состав OID функции из системной таблицы `pg_proc`. Таким образом, функциям Postgres Pro с одним именем и разными типами аргументов так же будут соответствовать различные процедуры Tcl. Это обычно остаётся незамеченным для программиста PL/Tcl, но может проявиться при отладке.

Глава 42. PL/Perl — процедурный язык Perl

PL/Perl — это загружаемый процедурный язык, позволяющий реализовывать функции Postgres Pro на *языке программирования Perl*.

Основным преимуществом PL/Perl является то, что он позволяет применять в сохранённых функциях множество функций и операторов «перемалывания строк», имеющихся в Perl. Разобрать сложные строки на языке Perl может быть гораздо проще, чем используя строковые функции и управляющие структуры в PL/pgSQL.

Чтобы установить PL/Perl в определённую базу данных, выполните команду `CREATE EXTENSION plperl`.

> **Подсказка**
>
> Если язык устанавливается в `template1`, он будет автоматически установлен во все создаваемые впоследствии базы данных.

> **Примечание**
>
> Пользователи, имеющие дело с исходным кодом, должны явно включить сборку PL/Perl в процессе установки. Пользователи двоичных пакетов могут найти PL/Perl в отдельном модуле.

42.1. Функции на PL/Perl и их аргументы

Чтобы создать функцию на языке PL/Perl, используйте стандартный синтаксис CREATE FUNCTION:

```
CREATE FUNCTION имя_функции (типы-аргументов) RETURNS тип-результата AS $$
    # Тело функции на PL/Perl
$$ LANGUAGE plperl;
```

Тело функции содержит обычный код Perl. Фактически, код обвязки PL/Perl помещает этот код в подпрограмму Perl. Функция PL/Perl вызывается в скалярном контексте, так что она не может вернуть список. Не скалярные значения (массивы, записи и множества) можно вернуть по ссылке, как описывается ниже.

PL/Perl также поддерживает анонимные блоки кода, которые выполняются оператором DO:

```
DO $$
    # Код PL/Perl
$$ LANGUAGE plperl;
```

Анонимный блок кода не принимает аргументы, а любое значение, которое он мог бы вернуть, отбрасывается. В остальном он работает подобно коду функции.

> **Примечание**
>
> Использовать вложенные именованные подпрограммы в Perl опасно, особенно если они обращаются к лексическим переменным в окружающей области. Так как функция PL/Perl оборачивается в подпрограмму, любая именованная функция внутри неё будет вложенной. Вообще гораздо безопаснее создавать анонимные подпрограммы и вызывать их по ссылке на код. Дополнительную информацию вы можете получить на странице руководства man perldiag, в описании ошибок `Variable "%s" will not stay shared` (Переменная "%s" не останется разделяемой) и `Variable "%s" is not available` (Переменная "%s" недоступна),

либо найти в Интернете по ключевым словам «perl nested named subroutine» (perl вложенная именованная подпрограмма).

Синтаксис команды `CREATE FUNCTION` требует, чтобы тело функции было записано как строковая константа. Обычно для этого удобнее всего заключать строковую константу в доллары (см. Подраздел 4.1.2.4). Если вы решите применять синтаксис спецпоследовательностей `E''`, вам придётся дублировать апострофы (`'`) и обратную косую черту (`\`) в теле функции (см. Подраздел 4.1.2.1).

Аргументы и результат обрабатываются как и в любой другой подпрограмме на Perl: аргументы передаются в `@_`, а результирующим значением будет указанное в `return` или полученное в последнем выражении, вычисленном в функции.

Например, функцию, возвращающую большее из двух целых чисел, можно определить так:

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    if ($_[0] > $_[1]) { return $_[0]; }
    return $_[1];
$$ LANGUAGE plperl;
```

Примечание

Аргументы будут преобразованы из кодировки базы данных в UTF-8 для использования в PL/Perl, а при выходе снова будут преобразованы из UTF-8 в кодировку базы данных.

Если функции передаётся NULL-значение SQL, значением аргумента в Perl станет «undefined». Показанное выше определение функции будет не очень хорошо обрабатывать значения NULL (в действительности они будут восприняты как нули). Мы могли бы добавить указание `STRICT` в это определение, чтобы Postgres Pro поступал немного разумнее: при передаче значения NULL функция вовсе не будет вызываться, будет сразу возвращён результат NULL. С другой стороны, мы могли бы проверить значения undefined в теле функции. Например, предположим, что нам нужна функция `perl_max`, которая с одним аргументом NULL и вторым аргументом не NULL должна возвращать не NULL, а второй аргумент:

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    my ($x, $y) = @_;
    if (not defined $x) {
        return undef if not defined $y;
        return $y;
    }
    return $x if not defined $y;
    return $x if $x > $y;
    return $y;
$$ LANGUAGE plperl;
```

Как показано выше, чтобы выдать значение SQL NULL, нужно вернуть значение undefined. Это можно сделать и в строгой, и в нестрогой функции.

Всё в аргументах функции, что не является ссылкой, является строкой, то есть стандартным для Postgres Pro внешним текстовым представлением соответствующего типа данных. В случае с обычными числовыми или текстовыми типами, Perl просто воспринимает их должным образом, и программист, как правило, может об этом не думать. Однако в более сложных случаях может потребоваться преобразовать аргумент в форму, подходящую для использования в Perl. Например, для преобразования типа `bytea` в двоичное значение можно использовать функцию `decode_bytea`.

Аналогично, значения, передаваемые в Postgres Pro, должны быть в формате внешнего текстового представления. Например, для подготовки двоичных данных к возврату в значении `bytea` можно воспользоваться функцией `encode_bytea`.

Perl может возвращать массивы PostgreSQL как ссылки на массивы Perl. Например, так:

```
CREATE OR REPLACE function returns_array()
RETURNS text[][] AS $$
    return [['a"b','c,d'],['e\\f','g']];
$$ LANGUAGE plperl;

select returns_array();
```

Perl передаёт массивы PostgreSQL как объект, сопоставленный с `PostgreSQL::InServer::ARRAY`. С этим объектом можно работать как со ссылкой на массив или строкой, что допускает обратную совместимость с кодом Perl, написанным для PostgreSQL версии до 9.1. Например:

```
CREATE OR REPLACE FUNCTION concat_array_elements(text[]) RETURNS TEXT AS $$
    my $arg = shift;
    my $result = "";
    return undef if (!defined $arg);

    # в качестве ссылки на массив
    for (@$arg) {
        $result .= $_;
    }

    # также работает со строкой
    $result .= $arg;

    return $result;
$$ LANGUAGE plperl;

SELECT concat_array_elements(ARRAY['PL','/','Perl']);
```

> **Примечание**
>
> Многомерные массивы представляются как ссылки на массивы меньшей размерности со ссылками — этот способ хорошо знаком каждому программисту на Perl.

Аргументы составного типа передаются функции как ссылки на хеши. Ключами хеша являются имена атрибутов составного типа. Например:

```
CREATE TABLE employee (
    name text,
    basesalary integer,
    bonus integer
);

CREATE FUNCTION empcomp(employee) RETURNS integer AS $$
    my ($emp) = @_;
    return $emp->{basesalary} + $emp->{bonus};
$$ LANGUAGE plperl;

SELECT name, empcomp(employee.*) FROM employee;
```

Функция на PL/Perl может вернуть результат составного типа, применяя тот же подход: возвратить ссылку на хеш с требуемыми атрибутами. Например, так:

```
CREATE TYPE testrowperl AS (f1 integer, f2 text, f3 text);

CREATE OR REPLACE FUNCTION perl_row() RETURNS testrowperl AS $$
    return {f2 => 'hello', f1 => 1, f3 => 'world'};
```

```
$$ LANGUAGE plperl;

SELECT * FROM perl_row();
```

Столбцы объявленного типа результата, отсутствующие в хеше, будут возвращены как значения NULL.

Функции на PL/Perl могут также возвращать множества со скалярными или составными типами. Обычно желательно возвращать результат по одной строке, чтобы сократить время подготовки с одной стороны, и чтобы не потребовалось накапливать весь набор данных в памяти, с другой. Это можно реализовать с помощью функции `return_next`, как показано ниже. Заметьте, что после последнего вызова `return_next`, нужно поместить `return` или (что лучше) `return undef`.

```
CREATE OR REPLACE FUNCTION perl_set_int(int)
RETURNS SETOF INTEGER AS $$
    foreach (0..$_[0]) {
        return_next($_);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set()
RETURNS SETOF testrowperl AS $$
    return_next({ f1 => 1, f2 => 'Hello', f3 => 'World' });
    return_next({ f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' });
    return_next({ f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' });
    return undef;
$$ LANGUAGE plperl;
```

Для небольших наборов данных можно также вернуть ссылку на массив, содержащий скаляры, ссылки на массивы, либо ссылки на хеши для простых типов, типов массивов и составных типов, соответственно. Ниже приведена пара простых примеров, показывающих, как возвратить весь набор данных в виде ссылки на массив:

```
CREATE OR REPLACE FUNCTION perl_set_int(int) RETURNS SETOF INTEGER AS $$
    return [0..$_[0]];
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set() RETURNS SETOF testrowperl AS $$
    return [
        { f1 => 1, f2 => 'Hello', f3 => 'World' },
        { f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' },
        { f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' }
    ];
$$ LANGUAGE plperl;

SELECT * FROM perl_set();
```

Если вы хотите использовать в своём коде `strict`, у вас есть несколько вариантов. Для временного глобального использования вы можете задать для `plperl.use_strict` значение true командой `SET`. Это повлияет на компилируемые впоследствии функции PL/Perl, но не на функции, уже скомпилированные в текущем сеансе. Для постоянного глобального использования вы можете присвоить параметру `plperl.use_strict` значение true в файле `postgresql.conf`.

Для постоянного использования strict в опредёлённых функциях вы можете просто написать:

```
use strict;
```

в начале тела этих функций.

Вы также можете использовать указания `feature` в `use`, если используете Perl версии 5.10.0 или новее.

42.2. Значения в PL/Perl

Значения аргументов, передаваемые в код функции PL/Perl, представляют собой просто входные аргументы, преобразованные в текстовый вид (так же, как при выводе оператором `SELECT`). И наоборот, команды `return` и `return_next` могут принять любую строку, соответствующую формату ввода для объявленного типа результата функции.

42.3. Встроенные функции

42.3.1. Обращение к базе данных из PL/Perl

Обращаться к самой базе данных из кода Perl можно, используя следующие функции:

`spi_exec_query(`*`запрос`* `[, `*`макс-строк`*`])`

`spi_exec_query` выполняет команду SQL и возвращает весь набор строк в виде ссылки на массив хешей. *Эту функцию следует использовать, только если вы знаете, что набор будет относительно небольшим.* Так выглядит пример запроса (`SELECT`) с дополнительно заданным максимальным числом строк:

```
$rv = spi_exec_query('SELECT * FROM my_table', 5);
```

Этот запрос возвращает не больше 5 строк из таблицы `my_table`. Если в `my_table` есть столбец `my_column`, получить его значение из строки `$i` результата можно следующим образом:

```
$foo = $rv->{rows}[$i]->{my_column};
```

Общее число строк, возвращённых запросом `SELECT`, можно получить так:

```
$nrows = $rv->{processed}
```

Так можно выполнить команду другого типа:

```
$query = "INSERT INTO my_table VALUES (1, 'test')";
$rv = spi_exec_query($query);
```

Затем можно получить статус команды (например, `SPI_OK_INSERT`) следующим образом:

```
$res = $rv->{status};
```

Чтобы получить число затронутых строк, выполните:

```
$nrows = $rv->{processed};
```

Полный пример:

```
CREATE TABLE test (
    i int,
    v varchar
);

INSERT INTO test (i, v) VALUES (1, 'first line');
INSERT INTO test (i, v) VALUES (2, 'second line');
INSERT INTO test (i, v) VALUES (3, 'third line');
INSERT INTO test (i, v) VALUES (4, 'immortal');

CREATE OR REPLACE FUNCTION test_munge() RETURNS SETOF test AS $$
    my $rv = spi_exec_query('select i, v from test;');
    my $status = $rv->{status};
    my $nrows = $rv->{processed};
```

```
        foreach my $rn (0 .. $nrows - 1) {
            my $row = $rv->{rows}[$rn];
            $row->{i} += 200 if defined($row->{i});
            $row->{v} =~ tr/A-Za-z/a-zA-Z/ if (defined($row->{v}));
            return_next($row);
        }
        return undef;
    $$ LANGUAGE plperl;

    SELECT * FROM test_munge();
```

`spi_query(`*команда*`)`
`spi_fetchrow(`*cursor*`)`
`spi_cursor_close(`*cursor*`)`

Функции `spi_query` и `spi_fetchrow` применяются в паре, когда набор строк может быть очень большим или когда нужно возвращать строки по мере их поступления. Функция `spi_fetchrow` работает *только* с `spi_query`. Следующий пример показывает, как использовать их вместе:

```
    CREATE TYPE foo_type AS (the_num INTEGER, the_text TEXT);

    CREATE OR REPLACE FUNCTION lotsa_md5 (INTEGER) RETURNS SETOF foo_type AS $$
        use Digest::MD5 qw(md5_hex);
        my $file = '/usr/share/dict/words';
        my $t = localtime;
        elog(NOTICE, "opening file $file at $t" );
        open my $fh, '<', $file # здесь мы обращаемся к файлу!
            or elog(ERROR, "cannot open $file for reading: $!");
        my @words = <$fh>;
        close $fh;
        $t = localtime;
        elog(NOTICE, "closed file $file at $t");
        chomp(@words);
        my $row;
        my $sth = spi_query("SELECT * FROM generate_series(1,$_[0]) AS b(a)");
        while (defined ($row = spi_fetchrow($sth))) {
            return_next({
                the_num => $row->{a},
                the_text => md5_hex($words[rand @words])
            });
        }
        return;
    $$ LANGUAGE plperlu;

    SELECT * from lotsa_md5(500);
```

Обычно вызов `spi_fetchrow` нужно повторять, пока не будет получен результат `undef`, показывающий, что все строки уже прочитаны. Курсор, возвращаемый функцией `spi_query`, автоматически освобождается, когда `spi_fetchrow` возвращает `undef`. Если вы не хотите читать все строки, освободите курсор, выполнив `spi_cursor_close`, чтобы не допустить утечки памяти.

`spi_prepare(`*команда*`, `*типы аргументов*`)`
`spi_query_prepared(`*план*`, `*аргументы*`)`
`spi_exec_prepared(`*план* `[, `*атрибуты*`], `*аргументы*`)`
`spi_freeplan(`*план*`)`

Функции `spi_prepare`, `spi_query_prepared`, `spi_exec_prepared` и `spi_freeplan` реализуют ту же функциональность, но для подготовленных запросов. Функция `spi_prepare` принимает строку запроса с нумерованными местозаполнителями аргументов ($1, $2 и т. д.) и список строк с типами аргументов:

```
$plan = spi_prepare('SELECT * FROM test WHERE id > $1 AND name = $2',
                                                    'INTEGER', 'TEXT');
```

План запроса, подготовленный вызовом `spi_prepare`, можно использовать вместо строки запроса либо в `spi_exec_prepared`, возвращающей тот же результат, что и `spi_exec_query`, либо в `spi_query_prepared`, возвращающей курсор так же, как `spi_query`, который затем можно передать в `spi_fetchrow`. В необязательном втором параметре `spi_exec_prepared` можно передать хеш с атрибутами; в настоящее время поддерживается только атрибут `limit`, задающий максимальное число строк, которое может вернуть запрос.

Подготовленные запросы хороши тем, что позволяют использовать единожды подготовленный план для неоднократного выполнения запроса. Когда план оказывается не нужен, его можно освободить, вызвав `spi_freeplan`:

```
CREATE OR REPLACE FUNCTION init() RETURNS VOID AS $$
        $_SHARED{my_plan} = spi_prepare('SELECT (now() + $1)::date AS now',
                                        'INTERVAL');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION add_time( INTERVAL ) RETURNS TEXT AS $$
        return spi_exec_prepared(
                $_SHARED{my_plan},
                $_[0]
        )->{rows}->[0]->{now};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION done() RETURNS VOID AS $$
        spi_freeplan( $_SHARED{my_plan});
        undef $_SHARED{my_plan};
$$ LANGUAGE plperl;

SELECT init();
SELECT add_time('1 day'), add_time('2 days'), add_time('3 days');
SELECT done();

  add_time  |  add_time  |  add_time
------------+------------+------------
 2005-12-10 | 2005-12-11 | 2005-12-12
```

Заметьте, что параметры для `spi_prepare` обозначаются как $1, $2, $3 и т. д., так что по возможности не записывайте строки запросов в двойных кавычках, чтобы не спровоцировать трудноуловимые ошибки.

Ещё один пример, иллюстрирующий использование необязательного параметра `spi_exec_prepared`:

```
CREATE TABLE hosts AS SELECT id, ('192.168.1.'||id)::inet AS address
                      FROM generate_series(1,3) AS id;

CREATE OR REPLACE FUNCTION init_hosts_query() RETURNS VOID AS $$
        $_SHARED{plan} = spi_prepare('SELECT * FROM hosts
                                      WHERE address << $1', 'inet');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION query_hosts(inet) RETURNS SETOF hosts AS $$
        return spi_exec_prepared(
                $_SHARED{plan},
                {limit => 2},
                $_[0]
        )->{rows};
```

```
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION release_hosts_query() RETURNS VOID AS $$
        spi_freeplan($_SHARED{plan});
        undef $_SHARED{plan};
$$ LANGUAGE plperl;

SELECT init_hosts_query();
SELECT query_hosts('192.168.1.0/30');
SELECT release_hosts_query();

    query_hosts
-----------------
 (1,192.168.1.1)
 (2,192.168.1.2)
(2 rows)
```

42.3.2. Вспомогательные функции в PL/Perl

`elog(`*уровень*`, `*сообщение*`)`

Выдаёт служебное сообщение или сообщение об ошибке. Возможные уровни сообщений: `DEBUG` (ОТЛАДКА), `LOG` (СООБЩЕНИЕ), `INFO` (ИНФОРМАЦИЯ), `NOTICE` (ЗАМЕЧАНИЕ), `WARNING` (ПРЕДУПРЕЖДЕНИЕ) и `ERROR` (ОШИБКА). С уровнем `ERROR` выдаётся ошибка; если она не перехватывается окружающим кодом Perl, она распространяется в вызывающий запрос, что приводит к прерыванию текущей транзакции или подтранзакции. По сути то же самое делает команда `die` языка Perl. При использовании других уровней происходит просто вывод сообщения с заданным уровнем важности. Будут ли сообщения определённого уровня передаваться клиенту и/или записываться в журнал, определяется конфигурационными параметрами log_min_messages и client_min_messages. За дополнительными сведениями обратитесь к Главе 18.

`quote_literal(`*строка*`)`

Оформляет переданную строку для использования в качестве текстовой строки в SQL-операторе. Включённые в неё апострофы и обратная косая черта при этом дублируются. Заметьте, что `quote_literal` возвращает undef, когда получает аргумент undef; если такие аргументы возможны, часто лучше использовать `quote_nullable`.

`quote_nullable(`*строка*`)`

Оформляет переданную строку для использования в качестве текстовой строки в SQL-операторе; либо, если поступает аргумент undef, возвращает строку «NULL» (без кавычек). Символы апостроф и обратная косая черта дублируются должным образом.

`quote_ident(`*строка*`)`

Оформляет переданную строку для использования в качестве идентификатора в SQL-операторе. При необходимости идентификатор заключается в кавычки (например, если он содержит символы, недопустимые в открытом виде, или буквы в разном регистре). Если переданная строка содержит кавычки, они дублируются.

`decode_bytea(`*строка*`)`

Возвращает неформатированные двоичные данные, представленные содержимым заданной строки, которая должна быть закодирована как `bytea`.

`encode_bytea(`*строка*`)`

Возвращает закодированные в виде `bytea` двоичные данные, содержащиеся в переданной строке.

`encode_array_literal(`*`массив`*`)`
`encode_array_literal(`*`массив`*`, `*`разделитель`*`)`

Возвращает содержимое указанного массива в виде строки в формате массива (см. Подраздел 8.15.2). Возвращает значение аргумента неизменённым, если это не ссылка не массив. Разделитель элементов в строке массива по умолчанию — «, » (если разделитель не определён или undef).

`encode_typed_literal(`*`значение`*`, `*`имя_типа`*`)`

Преобразует переменную Perl в значение типа данных, указанного во втором аргументе, и возвращает строковое представление этого значения. Корректно обрабатывает вложенные массивы и значения составных типов.

`encode_array_constructor(`*`массив`*`)`

Возвращает содержимое переданного массива в виде строки в формате конструктора массива (см. Подраздел 4.2.12). Отдельные значения заключаются в кавычки функцией `quote_nullable`. Возвращает значение аргумента, заключённое в кавычки функцией `quote_nullable`, если аргумент — не ссылка на массив.

`looks_like_number(`*`строка`*`)`

Возвращает значение true, если содержимое переданной строки похоже на число, по правилам Perl, и false в обратном случае. Возвращает undef для аргумента undef. Ведущие и замыкающие пробелы игнорируются. Строки `Inf` и `Infinity` считаются представляющими число (бесконечность).

`is_array_ref(`*`аргумент`*`)`

Возвращает значение true, если переданный аргумент можно воспринять как ссылку на массив, то есть это ссылка на `ARRAY` или `PostgreSQL::InServer::ARRAY`. В противном случае возвращает false.

42.4. Глобальные значения в PL/Perl

Вы можете использовать для хранения данных, включая ссылки на код, глобальный хеш `%_SHARED`. Эти данные будут сохраняться между вызовами функции на протяжении всего текущего сеанса.

Простой пример работы с разделяемыми данными:

```
CREATE OR REPLACE FUNCTION set_var(name text, val text) RETURNS text AS $$
    if ($_SHARED{$_[0]} = $_[1]) {
        return 'ok';
    } else {
        return "cannot set shared variable $_[0] to $_[1]";
    }
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION get_var(name text) RETURNS text AS $$
    return $_SHARED{$_[0]};
$$ LANGUAGE plperl;

SELECT set_var('sample', 'Hello, PL/Perl!  How''s tricks?');
SELECT get_var('sample');
```

Это чуть более сложный пример, в котором используется ссылка на код:

```
CREATE OR REPLACE FUNCTION myfuncs() RETURNS void AS $$
    $_SHARED{myquote} = sub {
        my $arg = shift;
        $arg =~ s/(['\\])/\\$1/g;
        return "'$arg'";
```

```
    };
$$ LANGUAGE plperl;

SELECT myfuncs(); /* инициализация функции */

/* Определение функции, использующей функцию заключения в кавычки */

CREATE OR REPLACE FUNCTION use_quote(TEXT) RETURNS text AS $$
    my $text_to_quote = shift;
    my $qfunc = $_SHARED{myquote};
    return &$qfunc($text_to_quote);
$$ LANGUAGE plperl;
```

(Код выше можно было бы упростить до однострочной команды `return $_SHARED{myquote}->($_[0]);` в ущерб читаемости.)

По соображениям безопасности, PL/Perl выполняет функции, вызываемые некоторой ролью SQL, в отдельном интерпретаторе Perl, выделенном для этой роли. Это предотвращает случайное или злонамеренное влияние одного пользователя на поведение функций PL/Perl другого пользователя. В каждом интерпретаторе будет своё значение переменной `%_SHARED` и собственное глобальное состояние. Таким образом, две функции PL/Perl будут разделять одно значение `%_SHARED`, только если они выполняются одной ролью SQL. В приложении, выполняющем код в одном сеансе с разными ролями SQL (вызывающем функции `SECURITY DEFINER`, использующем команду `SET ROLE` и т. д.) может понадобиться явно предпринять дополнительные меры, чтобы функции на PL/Perl могли разделять данные через `%_SHARED`. Для этого сначала установите для функций, которые должны взаимодействовать, одного владельца, а затем задайте для них свойство `SECURITY DEFINER`. Разумеется, при этом нужно позаботиться о том, чтобы эти функции не могли сделать ничего непредусмотренного.

42.5. Доверенный и недоверенный PL/Perl

Обычно PL/Perl устанавливается в базу данных как «доверенный» язык программирования с именем `plperl`. При этом в целях безопасности определённые операции в Perl запрещаются. Вообще говоря, запрещаются все операции, взаимодействующие с окружением. В том числе, это операции с файлами, `require` и `use` (для внешних модулей). Поэтому функции на PL/Perl, в отличие от функций на C, никаким образом не могут взаимодействовать с внутренними механизмами сервера баз данных или обращаться к операционной системе с правами серверного процесса. Вследствие этого, использовать этот язык можно разрешить любому непривилегированному пользователю баз данных.

В следующем примере показана функция, которая не будет работать, потому что операции с файловой системы запрещены по соображениям безопасности:

```
CREATE FUNCTION badfunc() RETURNS integer AS $$
    my $tmpfile = "/tmp/badfile";
    open my $fh, '>', $tmpfile
        or elog(ERROR, qq{could not open the file "$tmpfile": $!});
    print $fh "Testing writing to a file\n";
    close $fh or elog(ERROR, qq{could not close the file "$tmpfile": $!});
    return 1;
$$ LANGUAGE plperl;
```

Создать эту функцию не удастся, так как при проверке её правильности будет обнаружено использование запрещённого оператора.

Иногда возникает желание написать на Perl код, функциональность которого не будет ограничиваться. Например, может потребоваться функция на Perl, которая будет посылать почту. Для таких потребностей PL/Perl также можно установить как «недоверенный» язык (обычно его называют PL/PerlU). В этом случае будут доступны все возможности языка Perl. Устанавливая язык, укажите имя `plperlu`, чтобы выбрать недоверенную вариацию PL/Perl.

Автор функции на PL/PerlU должен позаботиться о том, чтобы эту функцию нельзя было использовать не по назначению, так как она может делать всё, что может пользователь с правами администратора баз данных. Заметьте, что СУБД позволяет создавать функции на недоверенных языках только суперпользователям базы данных.

Если показанная выше функция будет создана суперпользователем, и при этом будет выбран язык `plperlu`, она выполнится успешно.

Таким же образом, в анонимном блоке кода на Perl разрешены абсолютно любые операции, если в качестве языка вместо `plperl` выбирается `plperlu`, но выполнять этот код должен суперпользователь.

> **Примечание**
>
> Тогда как функции на PL/Perl исполняются отдельными интерпретаторами Perl для каждой роли SQL, все функции на PL/PerlU, вызываемые в рамках сеанса, исполняются в одном интерпретаторе Perl (отличном от тех, что исполняют функции PL/Perl). Благодаря этому, функции PL/PerlU могут свободно разделять общие данные, но между функциями PL/Perl и PL/PerlU взаимодействие невозможно.

> **Примечание**
>
> Perl поддерживает работу нескольких интерпретаторов в одном процессе, только если он был собран с нужными флагами, а именно, с флагом `usemultiplicity` или с флагом `useithreads`. (В отсутствие веских причин использовать потоки предпочтительным является вариант `usemultiplicity`. Дополнительную информацию вы можете получить на странице man perlembed.) При использовании PL/Perl с версией Perl, собранной без этих флагов, в рамках сеанса можно будет запустить только один интерпретатор Perl, так что в сеансе будет возможно выполнять либо функции PL/PerlU, либо функции PL/Perl (и вызывать их должна одна роль SQL).

42.6. Триггеры на PL/Perl

PL/Perl можно использовать для написания триггерных функций. В триггерной функции хеш-массив `$_TD` содержит информацию о произошедшем событии триггера. `$_TD` — глобальная переменная, которая получает нужное локальное значение при каждом вызове триггера. Хеш-массив `$_TD` содержит следующие поля:

`$_TD->{new}{foo}`

Новое значение столбца `foo`

`$_TD->{old}{foo}`

Старое значение столбца `foo`

`$_TD->{name}`

Имя вызываемого триггера

`$_TD->{event}`

Событие триггера: `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` или `UNKNOWN`

`$_TD->{when}`

Когда вызывается триггер: `BEFORE` (ДО), `AFTER` (ПОСЛЕ), `INSTEAD OF` (ВМЕСТО) или `UNKNOWN` (НЕИЗВЕСТНО)

`$_TD->{level}`

Уровень триггера: `ROW` (СТРОКА), `STATEMENT` (ОПЕРАТОР) или `UNKNOWN` (НЕИЗВЕСТНЫЙ)

`$_TD->{relid}`

OID таблицы, для которой сработал триггер

`$_TD->{table_name}`

Имя таблицы, для которой сработал триггер

`$_TD->{relname}`

Имя таблицы, для которой сработал триггер. Это обращение устарело и может быть ликвидировано в будущем выпуске. Используйте вместо него $_TD->{table_name}.

`$_TD->{table_schema}`

Имя схемы, содержащей таблицу, для которой сработал триггер

`$_TD->{argc}`

Число аргументов в триггерной функции

`@{$_TD->{args}}`

Аргументы триггерной функции. Не определено, если `$_TD->{argc}` равно 0.

В триггерах уровня строки возможны следующие варианты возврата:

`return;`

Выполнить операцию

`"SKIP"`

Не выполнять операцию

`"MODIFY"`

Указывает, что строка `NEW` была изменена триггерной функцией

Следующий пример триггерной функции иллюстрирует описанные выше варианты:

```
CREATE TABLE test (
    i int,
    v varchar
);

CREATE OR REPLACE FUNCTION valid_id() RETURNS trigger AS $$
    if (($_TD->{new}{i} >= 100) || ($_TD->{new}{i} <= 0)) {
        return "SKIP";    # пропустить команду INSERT/UPDATE
    } elsif ($_TD->{new}{v} ne "immortal") {
        $_TD->{new}{v} .= "(modified by trigger)";
        return "MODIFY";  # изменить строку и выполнить команду INSERT/UPDATE
    } else {
        return;           # выполнить команду INSERT/UPDATE
    }
$$ LANGUAGE plperl;

CREATE TRIGGER test_valid_id_trig
    BEFORE INSERT OR UPDATE ON test
    FOR EACH ROW EXECUTE PROCEDURE valid_id();
```

42.7. Событийные триггеры на PL/Perl

PL/Perl можно использовать для написания функций событийных триггеров. В функции событийного триггера хеш-массив `$_TD` содержит информацию о произошедшем событии триггера. `$_TD` — глобальная переменная, которая получает нужное локальное значение при каждом вызове триггера. Хеш-массив `$_TD` содержит следующие поля:

`$_TD->{event}`

Имя события, при котором срабатывает этот триггер.

`$_TD->{tag}`

Тег команды, для которой срабатывает этот триггер.

Возвращаемое значение триггерной процедуры игнорируется.

Следующий пример функции событийного триггера иллюстрирует описанное выше:

```
CREATE OR REPLACE FUNCTION perlsnitch() RETURNS event_trigger AS $$
  elog(NOTICE, "perlsnitch: " . $_TD->{event} . " " . $_TD->{tag} . " ");
$$ LANGUAGE plperl;

CREATE EVENT TRIGGER perl_a_snitch
    ON ddl_command_start
    EXECUTE PROCEDURE perlsnitch();
```

42.8. Внутренние особенности PL/Perl

42.8.1. Конфигурирование

В этом разделе описываются параметры конфигурации, влияющие на работу PL/Perl.

`plperl.on_init` (`string`)

Задаёт код Perl, который будет выполняться при первой инициализации интерпретатора Perl, до того, как он получает специализацию `plperl` или `plperlu`. Когда этот код выполняется, функции SPI ещё не доступны. Если выполнение кода завершается ошибкой, инициализация интерпретатора прерывается и ошибка распространяется в вызывающий запрос, в результате чего текущая транзакция или подтранзакция прерывается.

Размер этого кода ограничивается одной строкой. Более объёмный код можно поместить в модуль и загрузить этот модуль в строке `on_init`. Например:

```
plperl.on_init = 'require "plperlinit.pl"'
plperl.on_init = 'use lib "/my/app"; use MyApp::PgInit;'
```

Любые модули, загруженные в `plperl.on_init`, явно или неявно, будут доступны для использования в коде на языке `plperl`. Это может создать угрозу безопасности. Чтобы определить, какие модули были загружены, можно выполнить:

```
DO 'elog(WARNING, join ", ", sort keys %INC)' LANGUAGE plperl;
```

Если библиотека `plperl` включена в shared_preload_libraries, инициализация произойдёт в главном процессе (postmaster) и в этом случае необходимо очень серьёзно оценить риск нарушения работоспособности этого процесса. Основной смысл использовать эту возможность в том, чтобы модули Perl, подключаемые в `plperl.on_init`, загружались только при запуске главного процесса, и это исключало бы издержки загрузки для отдельных сеансов. Однако имейте в виду, что эти издержки исключаются только при загрузке в сеансе первого интерпретатора Perl — будь то PL/PerlU или PL/Perl для первой SQL-роли, вызывающей функцию на PL/Perl. Любые дополнительные интерпретаторы Perl, создаваемые в сеансе базы данных, должны будут выполнять `plperl.on_init` заново. Также учтите, что в Windows

предварительная загрузка не даёт никакого выигрыша, так как интерпретатор Perl, созданный в главном процессе, не передаётся дочерним процессам.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`plperl.on_plperl_init` (`string`)
`plperl.on_plperlu_init` (`string`)

В этих параметрах задаётся код Perl, который будет выполняться в момент, когда интерпретатор Perl получает специализацию `plperl` или `plperlu`, соответственно. Это произойдёт, когда в рамках сеанса будет первый раз вызвана функция на PL/Perl или PL/PerlU, либо когда потребуется дополнительный интерпретатор при использовании другого языка или при вызове функции PL/Perl новой SQL-ролью. Этот код выполняется после инициализации, произведённой в `plperl.on_init`. Однако функции SPI в момент исполнения этого кода ещё не доступны. Код в `plperl.on_plperl_init` запускается после того, как интерпретатор «помещается под замок», так что в нём разрешаются только доверенные операции.

Если этот код завершается ошибкой, инициализация прерывается и ошибка распространяется в вызывающий запрос, что приводит к прерыванию текущей транзакции или подтранзакции. При этом любые действия, уже произведённые в Perl, не будут отменены; однако использоваться этот интерпретатор больше не будет. При следующей попытке использовать этот язык система попытается заново инициализировать свежий интерпретатор Perl.

Изменять эти параметры разрешено только суперпользователям. Хотя изменить их можно в рамках сеанса, такие изменения не повлияют на работу интерпретаторов Perl, задействованных для выполнения функций ранее.

`plperl.use_strict` (`boolean`)

При значении, равном true, последующая компиляция функций PL/Perl будет выполняться с включённым указанием `strict`. Этот параметр не влияет на функции, уже скомпилированные в текущем сеансе.

42.8.2. Ограничения и недостающие возможности

Следующие возможности в настоящее время в PL/Perl отсутствуют, но их реализация будет желанной доработкой.

- Функции на PL/Perl не могут напрямую вызывать друг друга.
- SPI ещё не полностью реализован.
- Если вы выбираете очень большие наборы данных, используя `spi_exec_query`, вы должны понимать, что все эти данные загружаются в память. Вы можете избежать этого, используя пару функций `spi_query`/`spi_fetchrow`, как показано ранее.

  Похожая проблема возникает, если функция, возвращающая множество, передаёт в Postgres Pro большое число строк, выполняя `return`. Этой проблемы так же можно избежать, выполняя для каждой возвращаемой строки `return_next`, как показано ранее.
- Когда сеанс завершается штатно, не по причине критической ошибки, в Perl выполняются все блоки `END`, которые были определены. Никакие другие действия в настоящее время не выполняются. В частности, буферы файлов автоматически не сбрасываются и объекты автоматически не уничтожаются.

Глава 43. PL/Python — процедурный язык Python

Процедурный язык PL/Python позволяет писать функции Postgres Pro на *языке Python*.

Чтобы установить PL/Python в определённую базу данных, выполните команду `CREATE EXTENSION plpythonu` (но смотрите также Раздел 43.1).

> **Подсказка**
>
> Если язык устанавливается в `template1`, он будет автоматически установлен во все создаваемые впоследствии базы данных.

PL/Python представлен только в виде «недоверенного» языка, что означает, что он никаким способом не ограничивает действия пользователей, и поэтому он называется `plpythonu`. Доверенная вариация `plpython` может появиться в будущем, если в Python будет разработан безопасный механизм выполнения. Автор функции на недоверенном языке PL/Python должен позаботиться о том, чтобы эту функцию нельзя было использовать не по назначению, так как она может делать всё, что может пользователь с правами администратора баз данных. Создавать функции на недоверенных языках, таких как `plpythonu`, разрешено только суперпользователям.

> **Примечание**
>
> Пользователи, имеющие дело с исходным кодом, должны явно включить сборку PL/Python в процессе установки. (За дополнительными сведениями обратитесь к инструкциям по установке.) Пользователи двоичных пакетов могут найти PL/Python в отдельном модуле.

43.1. Python 2 и Python 3

PL/Python поддерживает две вариации языка: Python 2 и Python 3. (Более точная информация о поддерживаемых второстепенных версиях Python может содержаться в инструкциях по установке Postgres Pro.) Так как языки Python 2 и Python 3 несовместимы в некоторых важных аспектах, во избежание смешения их в PL/Python применяется следующая схема именования:

- Язык PostgreSQL с именем `plpython2u` представляет реализацию PL/Python, основанную на вариации языка Python 2.
- Язык PostgreSQL с именем `plpython3u` представляет реализацию PL/Python, основанную на вариации языка Python 3.
- Язык с именем `plpythonu` представляет реализацию PL/Python, основанную на версии Python по умолчанию, в данный момент это Python 2. (Этот выбор по умолчанию не зависит от того, какая версия считается локальной версией «по умолчанию», например, на какую версию указывает `/usr/bin/python`.) Выбор по умолчанию в отдалённом будущем выпуске Postgres Pro может быть сменён на Python 3, в зависимости от того, как будет происходить переход на Python 3 в сообществе Python.

Эта схема аналогична рекомендациям, данным в *PEP 394*, по выбору имени команды `python` и переходу с версии на версию.

Будет ли доступен PL/Python для Python 2 или для Python 3, либо сразу для обеих версий, зависит от конфигурации сборки или установленных пакетов.

> **Подсказка**
>
> Какая вариация будет собрана, зависит от того, как версия Python будет найдена при установке или будет задана в переменной окружения `PYTHON`. Чтобы в одной инсталляции

присутствовали обе вариации PL/Python, необходимо сконфигурировать и настроить дерево исходного кода дважды.

В результате формируется такая стратегия использования и смены определённой версии:

- Существующие пользователи и пользователи, которым в настоящее время неинтересен Python 3, могут выбрать имя языка `plpythonu` и им не придётся ничего менять в обозримом будущем. Чтобы упростить миграцию на Python 3, которая произойдёт в конце концов, рекомендуется постепенно проверять «готовность к будущему» кода, обновляя его до версий Python 2.6/2.7.

  На практике многие функции PL/Python можно мигрировать на Python 3 с минимальными изменениями или вовсе без изменений.

- Пользователи, знающие, что их код очень сильно зависит от Python 2, и не планирующие когда-либо менять его, могут использовать имя языка `plpython2u`. Это будет работать ещё очень и очень долго, пока в Postgres Pro не будет полностью ликвидирована поддержка Python 2.

- Пользователи, желающие погрузиться в Python 3, могут выбрать имя языка `plpython3u`, и их код будет работать всегда, по сегодняшним стандартам. В отдалённом будущем, когда версией по умолчанию может стать Python 3, цифру «3» из имени языка можно будет убрать из эстетических соображений.

- Смельчаки, желающие уже сегодня получить операционное окружение только с Python 3, могут модифицировать `pg_pltemplate`, чтобы имя `plpythonu` было равнозначно `plpython3u`, отдавая себе отчёт в том, что такая инсталляция будет несовместима с остальным миром.

Дополнительную информацию о переходе на Python 3 можно также найти в описании *Что нового в Python 3.0*.

Использовать PL/Python на базе Python 2 и PL/Python на базе Python 3 в одном сеансе нельзя, так как это приведёт к конфликту символов в динамических модулях, что может повлечь сбой серверного процесса Postgres Pro. В системе есть проверка, предотвращающая смешение основных версий Python в одном сеансе, которая прервёт сеанс при выявлении расхождения. Однако использовать обе вариации в одной базе данных всё же возможно, обращаясь к ним в разных сеансах.

43.2. Функции на PL/Python

Функции на PL/Python объявляются стандартным образом с помощью команды CREATE FUNCTION:

```
CREATE FUNCTION имя_функции (аргументы)
  RETURNS возвращаемый_тип
AS $$
  # Тело функции на PL/Python
$$ LANGUAGE plpythonu;
```

Тело функции содержит просто скрипт на языке Python. Когда вызывается функция, её аргументы передаются в виде элементов списка `args`; именованные аргументы также передаются скрипту Python как обычные переменные. С применением именованных аргументов скрипт обычно лучше читается. Результат из кода Python возвращается обычным способом, командой `return` или `yield` (в случае функции, возвращающей множество). Если возвращаемое значение не определено, Python возвращает `None`. Исполнитель PL/Python преобразует `None` языка Python в значение NULL языка SQL.

Например, функцию, возвращающее большее из двух целых чисел, можно определить так:

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
```

```
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

Код на Python, заданный в качестве тела объявляемой функции, становится телом функции Python. Например, для показанного выше объявления получается функция:

```
def __plpython_procedure_pymax_23456():
  if a > b:
    return a
  return b
```

Здесь 23456 — это OID, который Postgres Pro присвоил данной функции.

Значения аргументов задаются в глобальных переменных. Согласно правилам видимости в Python, тонким следствием этого является то, что переменной аргумента нельзя присвоить внутри функции выражение, включающее имя самой этой переменной, если только эта переменная не объявлена глобальной в текущем блоке. Например, следующий код не будет работать:

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  x = x.strip()  # ошибка
  return x
$$ LANGUAGE plpythonu;
```

так как присваивание `x` значения делает `x` локальной переменной для всего блока, и при этом `x` в правой части присваивания оказывается ещё не определённой локальной переменной `x`, а не параметром функции PL/Python. Добавив оператор `global`, это можно исправить:

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  global x
  x = x.strip()  # теперь всё в порядке
  return x
$$ LANGUAGE plpythonu;
```

Однако рекомендуется не полагаться на такие особенности реализации PL/Python, а принять, что параметры функции предназначены только для чтения.

43.3. Значения данных

Вообще говоря, цель исполнителя PL/Python — обеспечить «естественное» соответствие между мирами PostgreSQL и Python. Этим объясняется выбор правил сопоставления данных, описанных ниже.

43.3.1. Сопоставление типов данных

Когда вызывается функция PL/Python, её аргументы преобразуются из типа PostgreSQL в соответствующий тип Python по таким правилам:

- Тип PostgreSQL `boolean` преобразуется в тип `bool` языка Python.
- Типы PostgreSQL `smallint` и `int` преобразуются в тип `int` языка Python. Типы PostgreSQL `bigint` и `oid` становятся типами `long` в Python 2 и `int` в Python 3.
- Типы PostgreSQL `real` и `double` преобразуются в тип `float` языка Python.
- Тип PostgreSQL `numeric` преобразуется в тип `Decimal` среды Python. Этот тип импортируется из пакета `cdecimal`, при его наличии. В противном случае используется `decimal.Decimal` из стандартной библиотеки. Тип `cdecimal` работает значительно быстрее, чем `decimal`. Однако

в Python версии 3.3 и выше тип `cdecimal` включается в стандартную библиотеку под именем `decimal`, так что теперь этого различия нет.

- Тип PostgreSQL `bytea` становится типом `str` в Python 2 и `bytes` в Python 3. В Python 2 такую строку следует воспринимать как последовательность байт без какой-либо определённой кодировки символов.
- Все другие типы данных, включая типы символьных строк PostgreSQL, преобразуются в тип `str` языка Python. В Python 2 эта строка будет передаваться в кодировке сервера Postgres Pro; в Python 3 это будет строка в Unicode, как и все строки.
- Информация о нескалярных типах данных приведена ниже.

При завершении функции PL/Python её значение результата преобразуется в тип данных, объявленный как тип результата в PostgreSQL, следующим образом:

- Когда тип результата функции в PostgreSQL — `boolean`, возвращаемое значение приводится к логическому типу по правилам, принятым в *Python*. То есть false будет возвращено для 0 и пустой строки, но, обратите внимание, для `'f'` будет возвращено true.
- Когда тип результата функции PostgreSQL — `bytea`, возвращаемое значение будет преобразовано в строку (Python 2) или набор байт (Python 3), используя встроенные средства Python, а затем будет приведено к типу `bytea`.
- Для всех других типов результата PostgreSQL возвращаемое значение преобразуется в строку с помощью встроенной в Python функции `str`, и полученная строка передаётся функции ввода типа данных PostgreSQL. (Если значение в Python имеет тип `float`, оно преобразуется встроенной функцией `repr`, а не `str`, для недопущения потери точности.)

  Из кода Python 2 строки должны передаваться в Postgres Pro в кодировке сервера Postgres Pro. При передаче строки, неприемлемой для текущей кодировки сервера, возникает ошибка, но не все несоответствия кодировки могут быть выявлены, так что с некорректной кодировкой всё же могут быть получены нечитаемые строки. Строки Unicode переводятся в нужную кодировку автоматически, так что использовать их может быть безопаснее и удобнее. В Python 3 все строки имеют кодировку Unicode.
- Информация о нескалярных типах данных приведена ниже.

Заметьте, что логические несоответствия между объявленным в PostgreSQL типом результата и типом фактически возвращаемого объекта Python игнорируются — значение преобразуется в любом случае.

43.3.2. Null, None

Если функции передаётся значение SQL NULL, в Python значением этого аргумента будет `None`. Например, функция `pymax`, определённая как показано в Раздел 43.2, возвратит неверный ответ, получив аргументы NULL. Мы могли бы добавить указание `STRICT` в определение функции, чтобы Postgres Pro поступал немного разумнее: при передаче значения NULL функция вовсе не будет вызываться, будет сразу возвращён результат NULL. С другой стороны, мы могли бы проверить аргументы на NULL в теле функции:

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if (a is None) or (b is None):
    return None
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

Как показано выше, чтобы выдать из функции PL/Python значение SQL NULL, нужно вернуть значение `None`. Это можно сделать и в строгой, и в нестрогой функции.

43.3.3. Массивы, списки

Значения массивов SQL передаются в PL/Python в виде списка Python. Чтобы вернуть значение массива SQL из функции PL/Python, возвратите список Python:

```
CREATE FUNCTION return_arr()
  RETURNS int[]
AS $$
return [1, 2, 3, 4, 5]
$$ LANGUAGE plpythonu;

SELECT return_arr();
 return_arr
-------------
 {1,2,3,4,5}
(1 row)
```

Многомерные массивы передаются в PL/Python в виде вложенных списков Python. Например, двухмерный массив представляется как список списков. При передаче многомерного массива SQL из функции PL/Python необходимо, чтобы все внутренние списки на каждом уровне имели одинаковый размер. Например:

```
CREATE FUNCTION test_type_conversion_array_int4(x int4[]) RETURNS int4[] AS $$
plpy.info(x, type(x))
return x
$$ LANGUAGE plpythonu;

SELECT * FROM test_type_conversion_array_int4(ARRAY[[1,2,3],[4,5,6]]);
INFO:  ([[1, 2, 3], [4, 5, 6]], <type 'list'>)
 test_type_conversion_array_int4
---------------------------------
 {{1,2,3},{4,5,6}}
(1 row)
```

Другие последовательности Python, например кортежи, тоже принимаются для обратной совместимости с PostgreSQL версии 9.6 и ниже (где многомерные массивы не поддерживались). Однако они всегда воспринимаются как одномерные массивы, чтобы не возникало неоднозначности с составными типами. По этой же причине когда в многомерном массиве используется составной тип, он должен представляться как кортеж, а не список.

Учтите, что в Python и строки являются последовательностями, что может давать неожиданные эффекты, хорошо знакомые тем, кто программирует на Python:

```
CREATE FUNCTION return_str_arr()
  RETURNS varchar[]
AS $$
return "hello"
$$ LANGUAGE plpythonu;

SELECT return_str_arr();
 return_str_arr
----------------
 {h,e,l,l,o}
(1 row)
```

43.3.4. Составные типы

Аргументы составного типа передаются функции в виде сопоставлений Python. Именами элементов сопоставления являются атрибуты составного типа. Если атрибут в переданной строке имеет значение NULL, он передаётся в сопоставлении значением `None`. Пример работы с составным типом:

```
CREATE TABLE employee (
  name text,
  salary integer,
  age integer
);

CREATE FUNCTION overpaid (e employee)
  RETURNS boolean
AS $$
  if e["salary"] > 200000:
    return True
  if (e["age"] < 30) and (e["salary"] > 100000):
    return True
  return False
$$ LANGUAGE plpythonu;
```

Возвратить составной тип или строку таблицы из функции Python можно несколькими способами. В следующих примерах предполагается, что у нас объявлен тип:

```
CREATE TYPE named_value AS (
  name   text,
  value  integer
);
```

Результат этого типа можно вернуть как:

Последовательность (кортеж или список, но не множество, так как оно не индексируется)

В возвращаемых объектах последовательностей должно быть столько элементов, сколько полей в составном типе результата. Элемент с индексом 0 присваивается первому полю составного типа, с индексом 1 — второму и т. д. Например:

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  return ( name, value )
  # или альтернативный вариант, в виде списка: return [ name, value ]
$$ LANGUAGE plpythonu;
```

Чтобы выдать SQL NULL для какого-нибудь столбца, вставьте в соответствующую позицию `None`.

Когда возвращается массив составных значений, его нельзя представить в виде списка, так как невозможно однозначно определить, представляет ли список Python составной тип или ещё одну размерность массива.

Сопоставление (словарь)

Значение столбца результата получается из сопоставления, в котором ключом является имя столбца. Например:

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  return { "name": name, "value": value }
$$ LANGUAGE plpythonu;
```

Любые дополнительные пары ключ/значение в словаре игнорируются, а отсутствие нужных ключей считается ошибкой. Чтобы выдать SQL NULL для какого-нибудь столбца, вставьте `None` с именем соответствующего столбца в качестве ключа.

Объект (любой объект с методом `__getattr__`)

Объект передаётся аналогично сопоставлению. Пример:

```
CREATE FUNCTION make_pair (name text, value integer)
```

```
  RETURNS named_value
AS $$
  class named_value:
    def __init__ (self, n, v):
      self.name = n
      self.value = v
  return named_value(name, value)

  # или просто
  class nv: pass
  nv.name = name
  nv.value = value
  return nv
$$ LANGUAGE plpythonu;
```

Также поддерживаются функции с параметрами `OUT` (выходными). Например:

```
CREATE FUNCTION multiout_simple(OUT i integer, OUT j integer) AS $$
return (1, 2)
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple();
```

43.3.5. Функции, возвращающие множества

Функция PL/Python также может возвращать множества, содержащие скалярные и составные типы. Это можно осуществить разными способами, так как возвращаемый объект внутри превращается в итератор. В следующих примерах предполагается, что у нас есть составной тип:

```
CREATE TYPE greeting AS (
  how text,
  who text
);
```

Множество в качестве результата можно возвратить, применив:

Последовательность (кортеж, список, множество)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  # возвращает кортеж, содержащий списки в качестве составных типов
  # также будут работать и остальные комбинации
  return ( [ how, "World" ], [ how, "PostgreSQL" ], [ how, "PL/Python" ] )
$$ LANGUAGE plpythonu;
```

Итератор (любой объект, реализующий методы `__iter__` и `next`)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  class producer:
    def __init__ (self, how, who):
      self.how = how
      self.who = who
      self.ndx = -1

    def __iter__ (self):
      return self

    def next (self):
      self.ndx += 1
```

```
        if self.ndx == len(self.who):
          raise StopIteration
        return ( self.how, self.who[self.ndx] )

    return producer(how, [ "World", "PostgreSQL", "PL/Python" ])
$$ LANGUAGE plpythonu;
```

Генератор (`yield`)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  for who in [ "World", "PostgreSQL", "PL/Python" ]:
    yield ( how, who )
$$ LANGUAGE plpythonu;
```

Также поддерживаются функции, возвращающие множества, с параметрами `OUT` (объявленные с `RETURNS SETOF record`). Например:

```
CREATE FUNCTION multiout_simple_setof(n integer, OUT integer, OUT integer) RETURNS
 SETOF record AS $$
return [(1, 2)] * n
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple_setof(3);
```

43.4. Совместное использование данных

Для сохранения внутренних данных при повторных вызовах одной и той же функции предусмотрен глобальный словарь `SD`. Для размещения публичных данных предназначен глобальный словарь `GD`, доступный всем функциям на Python в сеансе; используйте его с осторожностью.

Каждая функция получает собственную среду выполнения в интерпретаторе Python, так что глобальные данные и аргументы функции, например `myfunc`, не будут доступны в `myfunc2`. Исключение составляют данные в словаре `GD`, как сказано выше.

43.5. Анонимные блоки кода

PL/Python также поддерживает анонимные блоки кода, которые выполняются оператором DO:

```
DO $$
    # Код на PL/Python
$$ LANGUAGE plpythonu;
```

Анонимный блок кода не принимает аргументы, а любое значение, которое он мог бы вернуть, отбрасывается. В остальном он работает подобно коду функции.

43.6. Триггерные функции

Когда функция используется как триггер, словарь `TD` содержит значения, связанные с работой триггера:

`TD["event"]`

содержит название события в виде строки: `INSERT`, `UPDATE`, `DELETE` или `TRUNCATE`.

`TD["when"]`

содержит одну из строк: `BEFORE`, `AFTER` или `INSTEAD OF`.

`TD["level"]`

содержит `ROW` или `STATEMENT`.

`TD["new"]`
`TD["old"]`

Для триггера уровня строки одно или оба этих поля содержат соответствующие строки триггера, в зависимости от события триггера.

`TD["name"]`

содержит имя триггера.

`TD["table_name"]`

содержит имя таблицы, для которой сработал триггер.

`TD["table_schema"]`

содержит схему таблицы, для которой сработал триггер.

`TD["relid"]`

содержит OID таблицы, для которой сработал триггер.

`TD["args"]`

Если в команде `CREATE TRIGGER` задавались аргументы, их можно получить как элементы массива с `TD["args"][0]` по `TD["args"][`*n*`-1]`.

Если в `TD["when"]` передано `BEFORE` или `INSTEAD OF`, а в `TD["level"]` — `ROW`, вы можете вернуть значение `None` или `"OK"` из функции Python, чтобы показать, что строка не была изменена, значение `"SKIP"`, чтобы прервать событие, либо, если в `TD["event"]` передана команда `INSERT` или `UPDATE`, вы можете вернуть `"MODIFY"`, чтобы показать, что новая строка была изменена. Во всех других случаях возвращаемое значение игнорируется.

43.7. Обращение к базе данных

Исполнитель языка PL/Python автоматически импортирует модуль Python с именем `plpy`. Вы в своём коде можете использовать функции и константы, объявленные в этом модуле, обращаясь к ним по именам вида `plpy.`*имя*.

43.7.1. Функции обращения к базе данных

Модуль `plpy` содержит различные функции для выполнения команд в базе данных:

`plpy.execute(`*запрос* `[, `*макс-строк*`])`

При вызове `plpy.execute` со строкой запроса и необязательным аргументом, ограничивающим число строк, выполняется заданный запрос, а то, что он выдаёт, возвращается в виде объекта результата.

Объект результата имитирует список или словарь. Получить из него данные можно по номеру строки и имени столбца. Например, команда:

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

вернёт не более 5 строк из отношения `my_table`. Если в `my_table` есть столбец `my_column`, к нему можно обратиться так:

```
foo = rv[i]["my_column"]
```

Число возвращённых в этом объекте строк можно получить, воспользовавшись встроенной функцией `len`.

Для объекта результата определены следующие дополнительные методы:

`nrows()`

Возвращает число строк, обработанных командой. Заметьте, что это число не обязательно будет равно числу возвращённых строк. Например, команда `UPDATE` устанавливает это значение, но не возвращает строк (без указания `RETURNING`).

`status()`

Значение состояния, возвращённое `SPI_execute()`.

`colnames()`
`coltypes()`
`coltypmods()`

Возвращают список имён столбцов, список OID типов столбцов и список модификаторов типа этих столбцов, соответственно.

Эти методы вызывают исключение, когда им передаётся объект, полученный от команды, не возвращающей результирующий набор, например, `UPDATE` без `RETURNING`, либо `DROP TABLE`. Но эти методы вполне можно использовать с результатом, содержащим ноль строк.

`__str__()`

Стандартный метод `__str__` определён так, чтобы можно было, например, вывести отладочное сообщение с результатами запроса, вызвав `plpy.debug(rv)`.

Объект результата может быть изменён.

Заметьте, что при вызове `plpy.execute` весь набор результатов будет прочитан в память. Эту функцию следует использовать, только если вы знаете, что набор будет относительно небольшим. Если вы хотите исключить риск переполнения памяти при выборке результатов большого объёма, используйте `plpy.cursor` вместо `plpy.execute`.

`plpy.prepare(`*`запрос`* `[, `*`типы_аргументов`*`])`
`plpy.execute(`*`план`* `[, `*`аргументы`* `[, `*`макс-строк`*`]])`

Функция `plpy.prepare` подготавливает план выполнения для запроса. Она вызывается со строкой запроса и списком типов параметров (если в запросе есть параметры). Например:

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE first_name = $1",
 ["text"])
```

Здесь `text` представляет переменную, передаваемую в качестве параметра `$1`. Второй аргумент необязателен, если запросу не нужно передавать никакие параметры.

Чтобы запустить подготовленный оператор на выполнение, используйте вариацию функции `plpy.execute`:

```
rv = plpy.execute(plan, ["name"], 5)
```

Передайте план в первом аргументе (вместо строки запроса), а список значений, которые будут подставлены в запрос, — во втором. Второй аргумент можно опустить, если запрос не принимает никакие параметры. Третий аргумент, как и раньше, задаёт необязательное ограничение максимального числа строк.

Вы также можете вызвать метод `execute` объекта плана:

```
rv = plan.execute(["name"], 5)
```

Параметры запросов и поля строк результата преобразуются между типами данных PostgreSQL и Python как описано в Разделе 43.3.

Когда вы подготавливаете план, используя модуль PL/Python, он сохраняется автоматически. Что это означает, вы можете узнать в документации SPI (Глава 44). Чтобы эффективно

использовать это в нескольких вызовах функции, может потребоваться применить словарь постоянного хранения `SD` или `GD` (см. Раздел 43.4). Например:

```
CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
    if "plan" in SD:
        plan = SD["plan"]
    else:
        plan = plpy.prepare("SELECT 1")
        SD["plan"] = plan
    # остальной код функции
$$ LANGUAGE plpythonu;
```

`plpy.cursor(`*запрос*`)`
`plpy.cursor(`*план* `[, `*аргументы*`])`

Функция `plpy.cursor` принимает те же аргументы, что и `plpy.execute` (кроме ограничения строк) и возвращает объект курсора, который позволяет обрабатывать объёмные наборы результатов небольшими порциями. Как и `plpy.execute`, этой функции можно передать строку запроса или объект плана со списком аргументов, а можно вызывать функцию `cursor` как метод объекта плана.

Объект курсора реализует метод `fetch`, который принимает целочисленный параметр и возвращает объект результата. При каждом следующем вызове `fetch` возвращаемый объект будет содержать следующий набор строк, в количестве, не превышающем значение параметра. Когда строки закончатся, `fetch` начнёт возвращать пустой объект результата. Объекты курсора также предоставляют *интерфейс итератора*, выдающий по строке за один раз, пока не будут выданы все строки. Данные, выбираемые таким образом, возвращаются не как объекты результата, а как словари (одной строке результата соответствует один словарь).

Следующий пример демонстрирует обработку содержимого большой таблицы двумя способами:

```
CREATE FUNCTION count_odd_iterator() RETURNS integer AS $$
odd = 0
for row in plpy.cursor("select num from largetable"):
    if row['num'] % 2:
         odd += 1
return odd
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_fetch(batch_size integer) RETURNS integer AS $$
odd = 0
cursor = plpy.cursor("select num from largetable")
while True:
    rows = cursor.fetch(batch_size)
    if not rows:
        break
    for row in rows:
        if row['num'] % 2:
            odd += 1
return odd
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_prepared() RETURNS integer AS $$
odd = 0
plan = plpy.prepare("select num from largetable where num % $1 <> 0", ["integer"])
rows = list(plpy.cursor(plan, [2]))  # или: = list(plan.cursor([2]))

return len(rows)
$$ LANGUAGE plpythonu;
```

Курсоры ликвидируются автоматически. Но если вы хотите явно освободить все ресурсы, занятые курсором, вызовите метод `close`. Продолжать получать данные через курсор, который был закрыт, нельзя.

> **Подсказка**
>
> Не путайте объекты, создаваемые функцией `plpy.cursor`, с курсорами DB-API, определёнными в *спецификации API для работы с базами данных в Python*. Они не имеют ничего общего, кроме имени.

43.7.2. Обработка ошибок

Функции, обращающиеся к базе данных, могут сталкиваться с ошибками, в результате которых они будут прерываться и вызывать исключение. Обе функции `plpy.execute` и `plpy.prepare` могут вызывать экземпляр подкласса исключения `plpy.SPIError`, которое по умолчание прекращает выполнение функции. Эту ошибку можно обработать, как и любое другое исключение в Python, применив конструкцию `try/except`. Например:

```
CREATE FUNCTION try_adding_joe() RETURNS text AS $$
    try:
        plpy.execute("INSERT INTO users(username) VALUES ('joe')")
    except plpy.SPIError:
        return "something went wrong"
    else:
        return "Joe added"
$$ LANGUAGE plpythonu;
```

Фактический класс вызываемого исключения соответствует определённому условию возникновения ошибки. Список всех возможных условий приведён в Таблице A.1. В модуле `plpy.spiexceptions` определяются классы исключений для каждого условия Postgres Pro, с именами, производными от имён условий. Например, имя `division_by_zero` становится именем `DivisionByZero`, `unique_violation` — именем `UniqueViolation`, `fdw_error` — именем `FdwError` и т. д. Все эти классы исключений наследуются от `SPIError`. Такое разделение на классы упрощает обработку определённых ошибок, например:

```
CREATE FUNCTION insert_fraction(numerator int, denominator int) RETURNS text AS $$
from plpy import spiexceptions
try:
    plan = plpy.prepare("INSERT INTO fractions (frac) VALUES ($1 / $2)", ["int",
 "int"])
    plpy.execute(plan, [numerator, denominator])
except spiexceptions.DivisionByZero:
    return "denominator cannot equal zero"
except spiexceptions.UniqueViolation:
    return "already have that fraction"
except plpy.SPIError, e:
    return "other error, SQLSTATE %s" % e.sqlstate
else:
    return "fraction inserted"
$$ LANGUAGE plpythonu;
```

Заметьте, что так как все исключения из модуля `plpy.spiexceptions` наследуются от исключения `SPIError`, команда `except`, обрабатывающая это исключение, будет перехватывать все ошибки при обращении к базе данных.

В качестве другого варианта обработки различных условий ошибок, вы можете перехватывать исключение `SPIError` и определять конкретное условие ошибки внутри блока `except` по

значению атрибута `sqlstate` объекта исключения. Этот атрибут содержит строку с кодом ошибки «SQLSTATE». Конечный результат при таком подходе примерно тот же.

43.8. Явные подтранзакции

Перехват ошибок, произошедших при обращении к базе данных, как описано в Подразделе 43.7.2, может привести к нежелательной ситуации, когда часть операций будет успешно выполнена, прежде чем произойдёт сбой. Данные останутся в несогласованном состоянии после обработки такой ошибки. PL/Python предлагает решение этой проблемы в форме явных подтранзакций.

43.8.1. Менеджеры контекста подтранзакций

Рассмотрим функцию, осуществляющую перевод средств между двумя счетами:

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
try:
    plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name =
 'joe'")
    plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name =
 'mary'")
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

Если при выполнении второго оператора `UPDATE` произойдёт исключение, эта функция сообщит об ошибке, но результат первого `UPDATE` будет тем не менее зафиксирован. Другими словами, средства будут списаны со счёта Джо, но не зачислятся на счёт Мэри.

Во избежание таких проблем вы можете завернуть вызовы `plpy.execute` в явную подтранзакцию. Модуль `plpy` предоставляет вспомогательный объект для управления явными подтранзакциями, создаваемый функцией `plpy.subtransaction()`. Объекты, созданные этой функцией, реализуют *интерфейс менеджера контекста*. Используя явные подтранзакции, мы можем переписать нашу функцию так:

```
CREATE FUNCTION transfer_funds2() RETURNS void AS $$
try:
    with plpy.subtransaction():
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name =
 'joe'")
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name =
 'mary'")
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

Заметьте, что конструкция `try/catch` по-прежнему нужна. Без неё исключение распространится вверх по стеку Python и приведёт к прерыванию всей функции с ошибкой Postgres Pro, так что в таблицу `operations` запись не добавится. Менеджер контекста подтранзакции не перехватывает ошибки, он только гарантирует, что все операции с базой данных в его области действия будут атомарно зафиксированы или отменены. Откат блока подтранзакции происходит при исключении любого вида, а не только исключения, вызванного ошибками при обращении к базе данных. Обычное исключение Python, вызванное внутри блока явной подтранзакции, также приведёт к откату этой подтранзакции.

43.8.2. Старые версии Python

Синтаксис использования менеджеров контекста с ключевым словом `with` по умолчанию поддерживается в Python 2.6. В PL/Python с более старой версией Python тоже возможно использовать явные подтранзакции, хотя и не так прозрачно. При этом вы можете вызывать методы `__enter__` и `__exit__` менеджера контекста по удобным псевдонимам `enter` и `exit`. Для такого случая функцию перечисления средств можно переписать так:

```
CREATE FUNCTION transfer_funds_old() RETURNS void AS $$
try:
    subxact = plpy.subtransaction()
    subxact.enter()
    try:
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name =
 'joe'")
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name =
 'mary'")
    except:
        import sys
        subxact.exit(*sys.exc_info())
        raise
    else:
        subxact.exit(None, None, None)
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"

plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

> **Примечание**
>
> Хотя менеджеры контекста были реализованы в 2.5, для использования синтаксиса `with` в этой версии нужно применить *«будущий оператор»*. Однако по техническим причинам «будущие операторы» в функциях PL/Python использовать нельзя.

43.9. Вспомогательные функции

Модуль `plpy` также предоставляет функции

```
plpy.debug( msg, **kwargs )
plpy.log( msg, **kwargs )
plpy.info( msg, **kwargs )
plpy.notice( msg, **kwargs )
plpy.warning( msg, **kwargs )
plpy.error( msg, **kwargs )
plpy.fatal( msg, **kwargs )
```

Функции `plpy.error` и `plpy.fatal` на самом деле выдают исключение Python, которое, если его не перехватить, распространяется в вызывающий запрос, что приводит к прерыванию текущей транзакции или подтранзакции. Команды `raise plpy.Error(msg)` и `raise plpy.Fatal(msg)` равнозначны вызовам `plpy.error(msg)` и `plpy.fatal(msg)`, соответственно, но форма `raise` не позволяет передавать аргументы с ключами. Другие функции просто выдают сообщения разных уровней важности. Будут ли сообщения определённого уровня передаваться клиентам и/или записываться в журнал сервера, определяется конфигурационными переменными log_min_messages и client_min_messages. За дополнительными сведениями обратитесь к Главе 18.

Аргумент *msg* задаётся как позиционный. Для обратной совместимости может быть передано несколько позиционных аргументов. В этом случае сообщением для клиента становится строковое представление кортежа позиционных аргументов.

Дополнительно только по ключам принимаются следующие аргументы:

```
detail
hint
sqlstate
schema_name
table_name
column_name
datatype_name
constraint_name
```

Строковое представление объектов, передаваемых в аргументах по ключам, позволяет выдать клиенту более богатую информацию. Например:

```
CREATE FUNCTION raise_custom_exception() RETURNS void AS $$
plpy.error("custom exception message",
           detail="some info about exception",
           hint="hint for users")
$$ LANGUAGE plpythonu;

=# SELECT raise_custom_exception();
ERROR:  plpy.Error: custom exception message
DETAIL:  some info about exception
HINT:  hint for users
CONTEXT:  Traceback (most recent call last):
  PL/Python function "raise_custom_exception", line 4, in <module>
    hint="hint for users")
PL/Python function "raise_custom_exception"
```

Ещё один набор вспомогательных функций образуют `plpy.quote_literal(`*строка*`)`, `plpy.quote_nullable(`*строка*`)` и `plpy.quote_ident(`*строка*`)`. Они равнозначны встроенным функциям заключения в кавычки, описанным в Разделе 9.4. Они полезны при конструировании свободно составляемых запросов. На PL/Python динамический SQL, показанный в Примере 40.1, формируется так:

```
plpy.execute("UPDATE tbl SET %s = %s WHERE key = %s" % (
    plpy.quote_ident(colname),
    plpy.quote_nullable(newvalue),
    plpy.quote_literal(keyvalue)))
```

43.10. Переменные окружения

Некоторые переменные окружения, воспринимаемые интерпретатором Python, тоже могут влиять на поведение PL/Python. При необходимости их нужно установить в среде основного серверного процесса Postgres Pro, например, в скрипте запуска. Множество доступных переменных окружения зависит от версии Python; за подробностями обратитесь к документации Python. На момент написания этой документации, на поведение PL/Python влияли следующие переменные окружения, при наличии подходящей версии Python:

- `PYTHONHOME`
- `PYTHONPATH`
- `PYTHONY2K`
- `PYTHONOPTIMIZE`
- `PYTHONDEBUG`

- PYTHONVERBOSE
- PYTHONCASEOK
- PYTHONDONTWRITEBYTECODE
- PYTHONIOENCODING
- PYTHONUSERBASE
- PYTHONHASHSEED

(Похоже, что вследствие тонкостей реализации Python, не зависящих от исполнителя PL/Python, некоторые переменные окружения, перечисленные на странице руководства man python, действуют только в интерпретаторе для командной строки, но не во встраиваемом интерпретаторе Python.)

Глава 44. Интерфейс программирования сервера

Интерфейс программирования сервера (SPI, Server Programming Interface) даёт разработчикам пользовательских функций на C возможность запускать команды SQL из своих функций. SPI представляет собой набор интерфейсных функций, упрощающих доступ к анализатору, планировщику и исполнителю запросов. В SPI есть также функции для управления памятью.

> **Примечание**
>
> Доступные процедурные языки предоставляют различные средства для выполнения SQL-команд из процедур. Большинство этих средств основаны на SPI, так что эта документация будет полезна и тем, кто использует эти языки.

Во избежание недопонимания мы будем употреблять слово «функция», говоря о функциях SPI, и слово «процедура», говоря о пользовательских функциях, написанных на C, и использующих SPI.

Учтите, что если команда, вызванная через SPI, прерывается ошибкой, управление не возвращается в вашу процедуру. Вместо этого происходит откат транзакции или подтранзакции, из которой вызывалась ваша процедура. (Это может показаться удивительным, с учётом того, что для большинства функций SPI описаны соглашения по возврату ошибок. Однако эти соглашения применимы только к ошибкам, выявляемым в самих функциях SPI.) Получить управление после ошибки можно, только организовав собственную подтранзакцию, окружающую вызовы SPI, в которых возможна ошибка.

Функции SPI выдают неотрицательный результат в случае успеха (либо через возвращаемое целочисленное значение, либо в глобальной переменной `SPI_result`, как описано ниже). В случае ошибки выдаётся отрицательный результат или `NULL`.

Файлы исходного кода, использующие SPI, должны включать заголовочный файл `executor/spi.h`.

44.1. Интерфейсные функции

SPI_connect

SPI_connect — подключить процедуру к менеджеру SPI

Синтаксис

```
int SPI_connect(void)
```

Описание

`SPI_connect` устанавливает подключение вызова процедуры к менеджеру SPI. Эту функцию необходимо вызвать, если вы хотите выполнять команды через SPI. Некоторые вспомогательные функции SPI могут вызываться из неподключённых процедур.

Возвращаемое значение

`SPI_OK_CONNECT`

при успехе

`SPI_ERROR_CONNECT`

при ошибке

SPI_finish

SPI_finish — отключить процедуру от менеджера SPI

Синтаксис

```
int SPI_finish(void)
```

Описание

`SPI_finish` закрывает текущее соединение с менеджером SPI. Эту функцию необходимо вызывать после завершения операций SPI, которые должны выполняться в текущем вызове процедуры. Однако если вы прерываете транзакцию, выполняя `elog(ERROR)`, о закрытии соединения можно не беспокоиться. В этом случае SPI произведёт очистку автоматически.

Возвращаемое значение

`SPI_OK_FINISH`

если отключение выполнено корректно

`SPI_ERROR_UNCONNECTED`

если вызывается из неподключённой процедуры

SPI_execute

SPI_execute — выполнить команду

Синтаксис

```
int SPI_execute(const char * command, bool read_only, long count)
```

Описание

`SPI_execute` выполняет заданную команду SQL для получения строк в количестве, ограниченном *`count`*. С параметром *`read_only`*, равным `true`, команда должна только читать данные; это несколько сокращает издержки на её выполнение.

Эту функцию можно вызывать только из подключённой процедуры.

Если *`count`* равен 0, команда выполняется для всех строк, к которым она применима. Если *`count`* больше нуля, будет получено не более чем *`count`* строк; выполнение команды остановится при достижении этого предела, практически так же, как и с предложением `LIMIT` в запросе. Например, команда:

```
SPI_execute("SELECT * FROM foo", true, 5);
```

получит из таблицы не более 5 строк. Заметьте, что это ограничение действует, только когда команда действительно возвращает строки. Например, эта команда:

```
SPI_execute("INSERT INTO foo SELECT * FROM bar", false, 5);
```

вставляет все строки из `bar`, игнорируя параметр *`count`*. Однако команда

```
SPI_execute("INSERT INTO foo SELECT * FROM bar RETURNING *", false, 5);
```

вставит не более 5 строк, так как её выполнение будет остановлено после получения пятой строки, выданной предложением `RETURNING`.

В одной строке можно передать несколько команд; `SPI_execute` возвращает результат команды, выполненной последней. Параметр *`count`* при этом будет применяться к каждой команде по отдельности (несмотря даже на то, что возвращён будет только последний результат). Это ограничение не будет распространяться на скрытые команды, генерируемые правилами.

Когда параметр *`read_only`* равен `false`, `SPI_execute` увеличивает счётчик команд и получает новый *снимок* перед выполнением каждой очередной команды в строке. Этот снимок фактически не меняется при текущем уровне изоляции транзакций `SERIALIZABLE` или `REPEATABLE READ`, но в режиме `READ COMMITTED` после обновления снимка очередная команда может видеть результаты только что зафиксированных транзакций из других сеансов. Это важно для согласованного поведения, когда команды модифицируют базу данных.

Когда параметр *`read_only`* равен `true`, `SPI_execute` не обновляет снимок и не увеличивает счётчик команд, и допускает в строке команд только `SELECT`. Заданные команды выполняются со снимком, ранее полученным для окружающего запроса. Этот режим выполнения несколько быстрее режима чтения/записи вследствие исключения издержек, связанных с отдельными командами. Он также позволяет создавать подлинно *стабильные* функции: так как последующие вызовы в транзакции будут использовать один снимок, результаты команд не изменятся.

Смешивать команды, только читающие, с командами, читающими и пишущими, в одной процедуре, использующей SPI, обычно неразумно; запросы только на чтение не увидят результатов изменений в базе данных, произведённых пишущими запросами.

Число строк, которые были фактически обработаны командой (последней), возвращается в глобальной переменной `SPI_processed`. Если эта функция возвращает значение `SPI_OK_SELECT`, `SPI_OK_INSERT_RETURNING`, `SPI_OK_DELETE_RETURNING` или `SPI_OK_UPDATE_RETURNING`, вы можете

обратиться по глобальному указателю SPITupleTable *SPI_tuptable и прочитать строки результата. Некоторые служебные команды (например, EXPLAIN) также возвращают наборы строк, и SPI_tuptable будет содержать их результаты и в этих случаях. Другие вспомогательные команды (COPY, CREATE TABLE AS) не возвращают набор строк, так что указатель SPI_tuptable равен NULL, но они так же возвращают число обработанных строк в SPI_processed.

Структура SPITupleTable определена так:

```
typedef struct
{
    MemoryContext tuptabcxt;    /* контекст таблицы результатов в памяти */
    uint64      alloced;        /* число зарезервированных в памяти значений */
    uint64      free;           /* число свободных значений */
    TupleDesc   tupdesc;        /* дескриптор строки */
    HeapTuple  *vals;           /* данные строк */
} SPITupleTable;
```

vals представляет собой массив указателей на строки. (Число записей в нём указывается в SPI_processed.) Поле tupdesc содержит дескриптор строки, который вы сможете передать функциям SPI, работающими со строками. Поля tuptabcxt, alloced и free предназначены для внутреннего использования, а не для процедур, работающих с SPI.

SPI_finish освобождает все структуры SPITupleTable, размещённые в памяти для текущей процедуры. Вы можете освободить структуру конкретной результирующей таблицы, если она вам не нужна, вызвав SPI_freetuptable.

Аргументы

const char * *command*

строка с командой, которая должна быть выполнена

bool *read_only*

true для режима выполнения «только чтение»

long *count*

максимальное число строк, которое должно быть возвращено; с 0 ограничения нет

Возвращаемое значение

Если команда была выполнена успешно, возвращается одно из следующих (неотрицательных) значений:

SPI_OK_SELECT

если выполнялась команда SELECT (но не SELECT INTO)

SPI_OK_SELINTO

если выполнялась команда SELECT INTO

SPI_OK_INSERT

если выполнялась команда INSERT

SPI_OK_DELETE

если выполнялась команда DELETE

SPI_OK_UPDATE

если выполнялась команда UPDATE

`SPI_OK_INSERT_RETURNING`

если выполнялась команда `INSERT RETURNING`

`SPI_OK_DELETE_RETURNING`

если выполнялась команда `DELETE RETURNING`

`SPI_OK_UPDATE_RETURNING`

если выполнялась команда `UPDATE RETURNING`

`SPI_OK_UTILITY`

если выполнялась служебная команда (например, `CREATE TABLE`)

`SPI_OK_REWRITTEN`

если команда была преобразована правилом в команду другого вида (например, `UPDATE` стал командой `INSERT`).

В случае ошибки возвращается одно из следующих отрицательных значений:

`SPI_ERROR_ARGUMENT`

если в качестве *`command`* передан `NULL` или *`count`* меньше 0

`SPI_ERROR_COPY`

при попытке выполнить `COPY TO stdout` или `COPY FROM stdin`

`SPI_ERROR_TRANSACTION`

при попытке выполнить команду управления транзакциями (`BEGIN`, `COMMIT`, `ROLLBACK`, `SAVEPOINT`, `PREPARE TRANSACTION`, `COMMIT PREPARED`, `ROLLBACK PREPARED` или любую их вариацию)

`SPI_ERROR_OPUNKNOWN`

если тип команды неизвестен (такого быть не должно)

`SPI_ERROR_UNCONNECTED`

если вызывается из неподключённой процедуры

Замечания

Все функции SPI, выполняющие запросы, заполняют и `SPI_processed`, и `SPI_tuptable` (только указатель, но не содержимое структуры). Сохраните эти две глобальные переменные в локальных переменных процедуры, если хотите обращаться к таблице результата `SPI_execute` или другой функции, выполняющей запрос, в нескольких вызовах процедуры.

SPI_exec

SPI_exec — выполнить команду чтения/записи

Синтаксис

```
int SPI_exec(const char * command, long count)
```

Описание

`SPI_exec` действует подобно `SPI_execute`, но ей не передаётся параметр *`read_only`* (всегда подразумевается `false`).

Аргументы

`const char * command`

строка с командой, которая должна быть выполнена

`long count`

максимальное число строк, которое должно быть возвращено; с `0` ограничения нет

Возвращаемое значение

См. `SPI_execute`.

SPI_execute_with_args

SPI_execute_with_args — выполнить команду с выделенными параметрами

Синтаксис

```
int SPI_execute_with_args(const char *command,
                          int nargs, Oid *argtypes,
                          Datum *values, const char *nulls,
                          bool read_only, long count)
```

Описание

`SPI_execute_with_args` выполняет команду, которая может включать ссылки на параметры, передаваемые извне. В тексте команды параметры обозначаются символами `$n`, а в вызове указываются типы данных и значения для каждого такого символа. Параметры *`read_only`* и *`count`* имеют тот же смысл, что и в `SPI_execute`.

Основное преимущество этой функции по сравнению с `SPI_execute` в том, что она позволяет передавать в команду значения данных, не требуя кропотливой подготовки строк, и таким образом сокращает риск атак с SQL-инъекцией.

Подобного результата можно достичь, вызвав `SPI_prepare` и затем `SPI_execute_plan`; однако с данной функцией план запроса всегда подстраивается под переданные конкретные значения параметров. Поэтому для разового выполнения запроса рекомендуется применять эту функцию. Если же одна и та же команда должна выполняться с самыми разными параметрами, какой вариант окажется быстрее, будет зависеть от стоимости повторного планирования и выигрыша от выбора специализированных планов.

Аргументы

`const char *` *`command`*

строка команды

`int` *`nargs`*

число входных параметров (`$1`, `$2` и т. д.)

`Oid *` *`argtypes`*

массив размера *`nargs`*, содержащий OID типов параметров

`Datum *` *`values`*

массив размера *`nargs`*, содержащий фактические значения параметров

`const char *` *`nulls`*

массив размера *`nargs`*, описывающий, в каких параметрах передаётся NULL

Если в *`nulls`* передаётся `NULL`, `SPI_execute_with_args` считает, что ни один из параметров не равен NULL. В противном случае элемент массива *`nulls`* должен содержать `' '`, если значение соответствующего параметра не NULL, либо `'n'`, если это значение — NULL. (В последнем случае значение, переданное в соответствующем элементе *`values`*, не учитывается.) Заметьте, что *`nulls`* — это не текстовая строка, а просто массив: ноль (`'\0'`) в конце не нужен.

`bool` *`read_only`*

`true` для режима выполнения «только чтение»

`long` *count*

максимальное число строк, которое должно быть возвращено; с `0` ограничения нет

Возвращаемое значение

Возвращаемые значения те же, что и у `SPI_execute`.

Переменные `SPI_processed` и `SPI_tuptable` устанавливаются как в `SPI_execute`, если вызов был успешным.

SPI_prepare

SPI_prepare — подготовить оператор, но пока не выполнять его

Синтаксис

```
SPIPlanPtr SPI_prepare(const char * command, int nargs, Oid * argtypes)
```

Описание

`SPI_prepare` создаёт и возвращает подготовленный оператор для заданной команды. Подготовленный оператор может быть затем неоднократно выполнен функцией `SPI_execute_plan`.

Когда одна и та же или похожие команды выполняются неоднократно, обычно выгоднее произвести анализ запроса только раз, а ещё выгоднее может быть повторно использовать план выполнения команды. `SPI_prepare` преобразует строку команды в подготовленный оператор, включающий в себя результаты анализа запроса. Подготовленный оператор также оставляет место для кеширования плана выполнения, если выбор специализированного плана для каждого выполнения не принесёт пользы.

Подготавливаемую команду можно сделать более общей, записав параметры (`$1`, `$2`, etc.) вместо значений, задаваемыми константами в обычной команде. Фактические значения параметров в этом случае будут задаваться при вызове `SPI_execute_plan`. Это позволяет применять подготовленную команду в более широком круге ситуаций, чем это возможно без параметров.

Оператор, возвращаемый функцией `SPI_prepare`, может использоваться только в текущем вызове процедуры, так как `SPI_finish` освобождает память, выделенную для такого оператора. Но этот оператор может быть сохранён на будущее с помощью функций `SPI_keepplan` или `SPI_saveplan`.

Аргументы

`const char * command`

строка команды

`int nargs`

число входных параметров (`$1`, `$2` и т. д.)

`Oid * argtypes`

указатель на массив, содержащий OID типов параметров

Возвращаемое значение

`SPI_prepare` возвращает ненулевой указатель на `SPIPlan`, скрытую структуру, представляющую подготовленный оператор. В случае ошибки возвращается `NULL`, а в `SPI_result` устанавливается один из кодов ошибок, определённых для `SPI_execute`, за исключением того, что код `SPI_ERROR_ARGUMENT` устанавливается, когда *`command`* — `NULL`, когда *`nargs`* меньше 0 или когда *`nargs`* больше 0, а *`argtypes`* — `NULL`.

Замечания

Если параметры не определены, при первом использовании `SPI_execute_plan` создаётся общий план, который затем будет применяться при последующих вызовах. Если же присутствуют параметры, `SPI_execute_plan` будет создавать специализированные планы для конкретных значений параметров. После достаточного количества использований полученного подготовленного оператора, функция `SPI_execute_plan` построит общий план, и если он не будет значительно дороже специализированных, она начнёт использовать его, а не будет строить план заново. Если это поведение по умолчанию не устраивает, его можно изменить,

передав флаг CURSOR_OPT_GENERIC_PLAN или CURSOR_OPT_CUSTOM_PLAN в SPI_prepare_cursor, чтобы ограничиться использованием только общего или специализированных планов, соответственно.

Хотя основной смысл подготовленного оператора в том, чтобы избежать повторного разбора и планирования запроса, Postgres Pro всё же будет принудительно повторять разбор и планирование запроса перед его выполнением, если со времени предыдущего использования подготовленного оператора произойдут изменения определений (DDL) объектов базы, задействованных в этом запросе. Также, если перед очередным использованием было изменено значение search_path, запрос будет разобран заново с новым значением search_path. (Последняя особенность появилась в PostgreSQL 9.3.) Чтобы узнать о поведении подготовленных операторов больше, обратитесь к PREPARE.

Эту функцию следует вызывать только из подключённой процедуры.

SPIPlanPtr объявлен в spi.h как указатель на скрытую структуру. Пытаться обращаться к её содержимому напрямую не стоит, так как ваш код скорее всего сломается при выходе новых версий Postgres Pro.

Имя SPIPlanPtr объясняется отчасти историческими причинами, так как теперь эта структура может не содержать собственно план выполнения.

SPI_prepare_cursor

SPI_prepare_cursor — подготовить оператор, но пока не выполнять его

Синтаксис

```
SPIPlanPtr SPI_prepare_cursor(const char * command, int nargs,
                              Oid * argtypes, int cursorOptions)
```

Описание

Функция `SPI_prepare_cursor` равнозначна `SPI_prepare`, за исключением того, что ей можно передать «параметры курсора». Эти параметры задаются битовой маской со значениями, определёнными в `nodes/parsenodes.h` для поля `options` структуры `DeclareCursorStmt`. `SPI_prepare` подразумевает, что эти параметры всегда нулевые.

Аргументы

`const char * command`

строка команды

`int nargs`

число входных параметров (`$1`, `$2` и т. д.)

`Oid * argtypes`

указатель на массив, содержащий OID типов параметров

`int cursorOptions`

битовая маска параметров курсора; 0 выбирает поведение по умолчанию

Возвращаемое значение

`SPI_prepare_cursor` возвращает результат по тем же соглашениям, что и `SPI_prepare`.

Замечания

К числу полезных бит, которые можно задать в *cursorOptions*, относятся `CURSOR_OPT_SCROLL`, `CURSOR_OPT_NO_SCROLL`, `CURSOR_OPT_FAST_PLAN`, `CURSOR_OPT_GENERIC_PLAN` и `CURSOR_OPT_CUSTOM_PLAN`. Заметьте, что параметр `CURSOR_OPT_HOLD` игнорируется.

SPI_prepare_params

SPI_prepare_params — подготовить оператор, но пока не выполнять его

Синтаксис

```
SPIPlanPtr SPI_prepare_params(const char * command,
                              ParserSetupHook parserSetup,
                              void * parserSetupArg,
                              int cursorOptions)
```

Описание

`SPI_prepare_params` создаёт и возвращает подготовленный оператор для заданной команды, но не выполняет саму команду. Эта функция равнозначна `SPI_prepare_cursor`, но позволяет вызывающему дополнительно установить функции-обработчики для управления разбором ссылок на внешние параметры.

Аргументы

`const char * command`

строка команды

`ParserSetupHook parserSetup`

Функция настройки обработчиков разбора

`void * parserSetupArg`

аргумент для сквозной передачи в *parserSetup*

`int cursorOptions`

битовая маска параметров курсора; 0 выбирает поведение по умолчанию

Возвращаемое значение

`SPI_prepare_params` возвращает результат по тем же соглашениям, что и `SPI_prepare`.

SPI_getargcount

SPI_getargcount — получить число аргументов, требующихся оператору, подготовленному функцией SPI_prepare

Синтаксис

```
int SPI_getargcount(SPIPlanPtr plan)
```

Описание

SPI_getargcount возвращает число аргументов, требующихся для выполнения оператора, подготовленного функцией SPI_prepare.

Аргументы

SPIPlanPtr *plan*

подготовленный оператор (возвращаемый функцией SPI_prepare)

Возвращаемое значение

Число аргументов, которое ожидает план, заданный параметром *plan*. Если значение *plan* неверное или NULL, в SPI_result устанавливается код SPI_ERROR_ARGUMENT, а функция возвращает -1.

SPI_getargtypeid

SPI_getargtypeid — получить OID типа аргумента для оператора, подготовленного функцией `SPI_prepare`

Синтаксис

```
Oid SPI_getargtypeid(SPIPlanPtr plan, int argIndex)
```

Описание

`SPI_getargtypeid` возвращает OID, представляющий тип аргумента под номером *`argIndex`* оператора, подготовленного функцией `SPI_prepare`. Первый аргумент идёт под номером ноль.

Аргументы

`SPIPlanPtr` *`plan`*

подготовленный оператор (возвращаемый функцией `SPI_prepare`)

`int` *`argIndex`*

индекс аргумента, начиная с нуля

Возвращаемое значение

OID типа аргумента с заданным индексом. Если значение *`plan`* неверное или `NULL`, либо *`argIndex`* меньше 0 или не меньше числа аргументов, объявленных при подготовке плана (передаваемого в *`plan`*), в `SPI_result` устанавливается `SPI_ERROR_ARGUMENT` и возвращается `InvalidOid`.

SPI_is_cursor_plan

SPI_is_cursor_plan — выдать `true`, если оператор, подготовленный функцией `SPI_prepare`, можно использовать с `SPI_cursor_open`

Синтаксис

```
bool SPI_is_cursor_plan(SPIPlanPtr plan)
```

Описание

`SPI_is_cursor_plan` возвращает `true`, если оператор, подготовленный функцией `SPI_prepare`, можно передать в качестве аргумента `SPI_cursor_open`, или `false` в противном случае. Для положительного ответа в *`plan`* должна быть представлена одна команда, и эта команда должна возвращать кортежи; например, `SELECT` может быть подходящей командой, если он не содержит предложения `INTO`, а `UPDATE` подходит, только если он содержит предложение `RETURNING`.

Аргументы

`SPIPlanPtr` *`plan`*

подготовленный оператор (возвращаемый функцией `SPI_prepare`)

Возвращаемое значение

Значение `true` или `false`, показывающее, можно ли для подготовленного оператора, заданного параметром *`plan`*, получить курсор, при `SPI_result` равном нулю. Если дать ответ невозможно (например, если значение *`plan`* неверное или `NULL`, либо вызывающий не подключён к SPI), в `SPI_result` устанавливается соответствующий код ошибки и возвращается `false`.

SPI_execute_plan

SPI_execute_plan — выполнить оператор, подготовленный функцией `SPI_prepare`

Синтаксис

```
int SPI_execute_plan(SPIPlanPtr plan, Datum * values, const char * nulls,
                     bool read_only, long count)
```

Описание

`SPI_execute_plan` выполняет оператор, подготовленный функцией `SPI_prepare` или родственными ей. Параметры *`read_only`* и *`count`* имеют тот же смысл, что и в `SPI_execute`.

Аргументы

`SPIPlanPtr` *`plan`*

подготовленный оператор (возвращаемый функцией `SPI_prepare`)

`Datum *` *`values`*

Массив фактических значений параметров. Его размер должен равняться числу аргументов оператора.

`const char *` *`nulls`*

Массив, описывающий, в каких параметрах передаётся NULL. Должен иметь размер, равный числу аргументов оператора.

Если в *`nulls`* передаётся `NULL`, `SPI_execute_plan` считает, что ни один из параметров не равен NULL. В противном случае элемент массива *`nulls`* должен содержать `' '`, если значение соответствующего параметра не NULL, либо `'n'`, если это значение — NULL. (В последнем случае значение, переданное в соответствующем элементе *`values`*, не учитывается.) Заметьте, что *`nulls`* — это не текстовая строка, а просто массив: ноль (`'\0'`) в конце не нужен.

`bool` *`read_only`*

`true` для режима выполнения «только чтение»

`long` *`count`*

максимальное число строк, которое должно быть возвращено; с `0` ограничения нет

Возвращаемое значение

Возвращаемые значения те же, что и у `SPI_execute`, со следующими дополнительными вариантами ошибок (отрицательных результатов):

`SPI_ERROR_ARGUMENT`

Если *`plan`* неверный или `NULL`, либо *`count`* меньше 0

`SPI_ERROR_PARAM`

Если в *`values`* передан `NULL` и *`plan`* был подготовлен с другими параметрами

Переменные `SPI_processed` и `SPI_tuptable` устанавливаются как в `SPI_execute`, если вызов был успешным.

SPI_execute_plan_with_paramlist

SPI_execute_plan_with_paramlist — выполнить оператор, подготовленный функцией `SPI_prepare`

Синтаксис

```
int SPI_execute_plan_with_paramlist(SPIPlanPtr plan,
                                    ParamListInfo params,
                                    bool read_only,
                                    long count)
```

Описание

`SPI_execute_plan_with_paramlist` выполняет оператор, подготовленный функцией `SPI_prepare`. Данная функция равнозначна `SPI_execute_plan`, не считая того, что информация о значениях параметров, передаваемых запросу, представляется по-другому. Представление `ParamListInfo` может быть удобным для передачи значений, уже имеющих нужный формат. Эта функция также поддерживает динамические наборы параметров, которые реализуются через функции-обработчики, устанавливаемые в `ParamListInfo`.

Аргументы

`SPIPlanPtr` *`plan`*

подготовленный оператор (возвращаемый функцией `SPI_prepare`)

`ParamListInfo` *`params`*

структура данных, содержащая типы и значения параметров; NULL, если их нет

`bool` *`read_only`*

`true` для режима выполнения «только чтение»

`long` *`count`*

максимальное число строк, которое должно быть возвращено; с `0` ограничения нет

Возвращаемое значение

Возвращаемые значения те же, что и у `SPI_execute_plan`.

Переменные `SPI_processed` и `SPI_tuptable` устанавливаются как в `SPI_execute_plan`, если вызов был успешным.

SPI_execp

SPI_execp — выполнить оператор в режиме чтения/записи

Синтаксис

```
int SPI_execp(SPIPlanPtr plan, Datum * values, const char * nulls, long count)
```

Описание

`SPI_execp` действует подобно `SPI_execute_plan`, но ей не передаётся параметр *`read_only`* (всегда подразумевается `false`).

Аргументы

`SPIPlanPtr` *`plan`*

подготовленный оператор (возвращаемый функцией `SPI_prepare`)

`Datum *` *`values`*

Массив фактических значений параметров. Его размер должен равняться числу аргументов оператора.

`const char *` *`nulls`*

Массив, описывающий, в каких параметрах передаётся NULL. Должен иметь размер, равный числу аргументов оператора.

Если в *`nulls`* передаётся `NULL`, `SPI_execp` считает, что ни один из параметров не равен NULL. В противном случае элемент массива *`nulls`* должен содержать `' '`, если значение соответствующего параметра не NULL, либо `'n'`, если это значение — NULL. (В последнем случае значение, переданное в соответствующем элементе *`values`*, не учитывается.) Заметьте, что *`nulls`* — это не текстовая строка, а просто массив: ноль (`'\0'`) в конце не нужен.

`long` *`count`*

максимальное число строк, которое должно быть возвращено; с `0` ограничения нет

Возвращаемое значение

См. `SPI_execute_plan`.

Переменные `SPI_processed` и `SPI_tuptable` устанавливаются как в `SPI_execute`, если вызов был успешным.

SPI_cursor_open

SPI_cursor_open — открыть курсор для оператора, созданного функцией SPI_prepare

Синтаксис

```
Portal SPI_cursor_open(const char * name, SPIPlanPtr plan,
                       Datum * values, const char * nulls,
                       bool read_only)
```

Описание

SPI_cursor_open открывает курсор (внутри называемый порталом), через который будет выполняться оператор, подготовленный функцией SPI_prepare. Параметры этой функции имеют тот же смысл, что и соответствующие параметры SPI_execute_plan.

Применение курсора по сравнению с непосредственным выполнением оператора даёт двойную выгоду. Во-первых, строки результата можно получать в небольших количествах, без риска исчерпать всю память при выполнении запросов, возвращающих много строк. Во-вторых, портал может существовать и после завершения текущей процедуры (на самом деле он может просуществовать до конца текущей транзакции). Возвратив имя портала в код, вызывающий процедуру, можно организовать выдачу результата в виде набора строк.

Переданные значения параметров копируются в портал курсора, так что их можно освободить и во время существования курсора.

Аргументы

`const char * name`

имя портала, либо NULL, чтобы имя выбрала система

`SPIPlanPtr plan`

подготовленный оператор (возвращаемый функцией SPI_prepare)

`Datum * values`

Массив фактических значений параметров. Его размер должен равняться числу аргументов оператора.

`const char * nulls`

Массив, описывающий, в каких параметрах передаётся NULL. Должен иметь размер, равный числу аргументов оператора.

Если в *nulls* передаётся NULL, SPI_cursor_open считает, что ни один из параметров не равен NULL. В противном случае элемент массива *nulls* должен содержать `' '`, если значение соответствующего параметра не NULL, либо `'n'`, если это значение — NULL. (В последнем случае значение, переданное в соответствующем элементе *values*, не учитывается.) Заметьте, что *nulls* — это не текстовая строка, а просто массив: ноль (`'\0'`) в конце не нужен.

`bool read_only`

true для режима выполнения «только чтение»

Возвращаемое значение

Указатель на портал, содержащий курсор. Заметьте, что соглашение о возврате ошибок отсутствует; все ошибки выдаются через elog.

SPI_cursor_open_with_args

SPI_cursor_open_with_args — открывает курсор для запроса с параметрами

Синтаксис

```
Portal SPI_cursor_open_with_args(const char *name,
                                 const char *command,
                                 int nargs, Oid *argtypes,
                                 Datum *values, const char *nulls,
                                 bool read_only, int cursorOptions)
```

Описание

`SPI_cursor_open_with_args` открывает курсор (внутри называемый порталом) для выполнения заданного запроса. Большинство параметров имеют тот же смысл, что и соответствующие параметры функций `SPI_prepare_cursor` и `SPI_cursor_open`.

Для разового выполнения запроса эту функцию следует предпочесть `SPI_prepare_cursor` с последующей `SPI_cursor_open`. Если же одна и та же команда должна выполняться с самыми разными параметрами, какой вариант окажется быстрее, будет зависеть от стоимости повторного планирования и выигрыша от выбора специализированных планов.

Переданные значения параметров копируются в портал курсора, так что их можно освободить и во время существования курсора.

Аргументы

`const char * name`

имя портала, либо `NULL`, чтобы имя выбрала система

`const char * command`

строка команды

`int nargs`

число входных параметров (`$1`, `$2` и т. д.)

`Oid * argtypes`

массив размера *`nargs`*, содержащий OID типов параметров

`Datum * values`

массив размера *`nargs`*, содержащий фактические значения параметров

`const char * nulls`

массив размера *`nargs`*, описывающий, в каких параметрах передаётся NULL

Если в *`nulls`* передаётся `NULL`, `SPI_cursor_open_with_args` считает, что ни один из параметров не равен NULL. В противном случае элемент массива *`nulls`* должен содержать `' '`, если значение соответствующего параметра не NULL, либо `'n'`, если это значение — NULL. (В последнем случае значение, переданное в соответствующем элементе *`values`*, не учитывается.) Заметьте, что *`nulls`* — это не текстовая строка, а просто массив: ноль (`'\0'`) в конце не нужен.

`bool read_only`

`true` для режима выполнения «только чтение»

`int` *`cursorOptions`*

битовая маска параметров курсора; 0 выбирает поведение по умолчанию

Возвращаемое значение

Указатель на портал, содержащий курсор. Заметьте, что соглашение о возврате ошибок отсутствует; все ошибки выдаются через `elog`.

SPI_cursor_open_with_paramlist

SPI_cursor_open_with_paramlist — открыть курсор с параметрами

Синтаксис

```
Portal SPI_cursor_open_with_paramlist(const char *name,
                                      SPIPlanPtr plan,
                                      ParamListInfo params,
                                      bool read_only)
```

Описание

`SPI_cursor_open_with_paramlist` открывает курсор (внутри называемый порталом) для выполнения оператора, подготовленного функцией `SPI_prepare`. Эта функция равнозначна `SPI_cursor_open`, не считая того, что информация о значениях параметров, передаваемых запросу, представляется по-другому. Представление `ParamListInfo` может быть удобным для передачи значений, уже имеющих нужный формат. Эта функция также поддерживает динамические наборы параметров через функции-обработчики, устанавливаемые в `ParamListInfo`.

Переданные значения параметров копируются в портал курсора, так что их можно освободить и во время существования курсора.

Аргументы

`const char * name`

имя портала, либо `NULL`, чтобы имя выбрала система

`SPIPlanPtr plan`

подготовленный оператор (возвращаемый функцией `SPI_prepare`)

`ParamListInfo params`

структура данных, содержащая типы и значения параметров; NULL, если их нет

`bool read_only`

`true` для режима выполнения «только чтение»

Возвращаемое значение

Указатель на портал, содержащий курсор. Заметьте, что соглашение о возврате ошибок отсутствует; все ошибки выдаются через `elog`.

SPI_cursor_find

SPI_cursor_find — найти существующий курсор по имени

Синтаксис

```
Portal SPI_cursor_find(const char * name)
```

Описание

`SPI_cursor_find` находит существующий портал по имени. В основном это полезно для разрешения имени курсора, возвращённого в текстовом виде какой-то другой функцией.

Аргументы

`const char * name`

имя портала

Возвращаемое значение

указатель на портал с заданным именем или `NULL`, если такой портал не найден

SPI_cursor_fetch

SPI_cursor_fetch — выбрать строки через курсор

Синтаксис

```
void SPI_cursor_fetch(Portal portal, bool forward, long count)
```

Описание

`SPI_cursor_fetch` выбирает некоторое количество строк через курсор. Эта функция реализует подмножество возможностей SQL-команды `FETCH` (расширенную функциональность предоставляет `SPI_scroll_cursor_fetch`).

Аргументы

`Portal` *`portal`*

портал, содержащий курсор

`bool` *`forward`*

true для выборки с перемещением вперёд, false — назад

`long` *`count`*

максимальное число строк, которое нужно выбрать

Возвращаемое значение

Переменные `SPI_processed` и `SPI_tuptable` устанавливаются как в `SPI_execute`, если вызов был успешным.

Замечания

Выборка назад может не поддерживаться, если план курсора был создан без параметра `CURSOR_OPT_SCROLL`.

SPI_cursor_move

SPI_cursor_move — переместить курсор

Синтаксис

```
void SPI_cursor_move(Portal portal, bool forward, long count)
```

Описание

`SPI_cursor_move` перемещает курсор на несколько строк. Эта функция реализует подмножество возможностей SQL-команды `MOVE` (расширенную функциональность предоставляет `SPI_scroll_cursor_move`).

Аргументы

`Portal` *`portal`*

портал, содержащий курсор

`bool` *`forward`*

true для перемещения вперёд, false — назад

`long` *`count`*

максимальное число строк, на какое возможно перемещение

Замечания

Перемещение назад может не поддерживаться, если план курсора был создан без параметра `CURSOR_OPT_SCROLL`.

SPI_scroll_cursor_fetch

SPI_scroll_cursor_fetch — выбрать строки через курсор

Синтаксис

```
void SPI_scroll_cursor_fetch(Portal portal, FetchDirection direction,
                             long count)
```

Описание

`SPI_scroll_cursor_fetch` выбирает некоторое количество строк через курсор. Её функциональность равнозначна `FETCH` в SQL.

Аргументы

`Portal` *`portal`*

портал, содержащий курсор

`FetchDirection` *`direction`*

один из вариантов: `FETCH_FORWARD`, `FETCH_BACKWARD`, `FETCH_ABSOLUTE` или `FETCH_RELATIVE`

`long` *`count`*

число строк, выбираемых с направлением `FETCH_FORWARD` или `FETCH_BACKWARD`; абсолютный номер выбираемой строки с вариантом `FETCH_ABSOLUTE`; либо относительный номер выбираемой строки с вариантом `FETCH_RELATIVE`

Возвращаемое значение

Переменные `SPI_processed` и `SPI_tuptable` устанавливаются как в `SPI_execute`, если вызов был успешным.

Замечания

Подробнее о параметрах *`direction`* и *`count`* рассказывается в описании SQL-команды FETCH.

Варианты направления, отличные от `FETCH_FORWARD`, могут не поддерживаться, если план курсора был создан без параметра `CURSOR_OPT_SCROLL`.

SPI_scroll_cursor_move

SPI_scroll_cursor_move — переместить курсор

Синтаксис

```
void SPI_scroll_cursor_move(Portal portal, FetchDirection direction,
                            long count)
```

Описание

`SPI_scroll_cursor_move` перемещает курсор на несколько строк. Её функциональность равнозначна `MOVE` в SQL.

Аргументы

`Portal` *`portal`*

портал, содержащий курсор

`FetchDirection` *`direction`*

один из вариантов: `FETCH_FORWARD`, `FETCH_BACKWARD`, `FETCH_ABSOLUTE` или `FETCH_RELATIVE`

`long` *`count`*

число строк, на которое сдвигается курсор, с направлением `FETCH_FORWARD` или `FETCH_BACKWARD`; абсолютный номер строки, к которой переходит курсор, с направлением `FETCH_ABSOLUTE`; либо относительный номер строки, к которой переходит курсор, с направлением `FETCH_RELATIVE`

Возвращаемое значение

В случае успеха переменная `SPI_processed` устанавливается как в `SPI_execute`. В `SPI_tuptable` оказывается `NULL`, так как эта функция не возвращает никакие строки.

Замечания

Подробнее о параметрах *`direction`* и *`count`* рассказывается в описании SQL-команды FETCH.

Варианты направления, отличные от `FETCH_FORWARD`, могут не поддерживаться, если план курсора был создан без параметра `CURSOR_OPT_SCROLL`.

SPI_cursor_close

SPI_cursor_close — закрыть курсор

Синтаксис

```
void SPI_cursor_close(Portal portal)
```

Описание

`SPI_cursor_close` закрывает ранее созданный курсор и освобождает память, занятую его порталом.

Все открытые курсоры закрываются автоматически в конце транзакции. Вызывать `SPI_cursor_close` может потребоваться, только если возникает желание освободить ресурсы скорее.

Аргументы

`Portal` *`portal`*

портал, содержащий курсор

SPI_keepplan

SPI_keepplan — сохранить подготовленный оператор

Синтаксис

```
int SPI_keepplan(SPIPlanPtr plan)
```

Описание

`SPI_keepplan` закрепляет переданный оператор (подготовленный функцией `SPI_prepare`), чтобы он не был ликвидирован функцией `SPI_finish` или диспетчером транзакций. Это даёт возможность повторно использовать подготовленные операторы при последующих вызовах вашей процедуры в текущем сеансе.

Аргументы

`SPIPlanPtr` *`plan`*

подготовленный оператор, который нужно сохранить

Возвращаемое значение

0 в случае успеха; `SPI_ERROR_ARGUMENT`, если *`plan`* неверный или `NULL`

Замечания

Переданный оператор перемещается в постоянное хранилище путём смены указателя (копировать данные не требуется). Если позже вы захотите удалить его, выполните для него `SPI_freeplan`.

SPI_saveplan

SPI_saveplan — сохранить подготовленный оператор

Синтаксис

```
SPIPlanPtr SPI_saveplan(SPIPlanPtr plan)
```

Описание

`SPI_saveplan` копирует переданный оператор (подготовленный функцией `SPI_prepare`) в память, чтобы он не был ликвидирован функцией `SPI_finish` или менеджером транзакций, и возвращает указатель на скопированный оператор. Это даёт возможность повторно использовать подготовленные операторы при последующих вызовах вашей процедуры в текущем сеансе.

Аргументы

`SPIPlanPtr plan`

подготовленный оператор, который нужно сохранить

Возвращаемое значение

Указатель на скопированный оператор, либо `NULL` в случае ошибки. При ошибке `SPI_result` принимает одно из этих значений:

`SPI_ERROR_ARGUMENT`

если *plan* неверный или `NULL`

`SPI_ERROR_UNCONNECTED`

если вызывается из неподключённой процедуры

Замечания

Изначально переданный оператор не освобождается, поэтому вы можете выполнить `SPI_freeplan` для него, чтобы высвободить память до `SPI_finish`.

В большинстве случаев `SPI_keepplan` предпочтительнее данной функции, так как она даёт примерно тот же результат, но обходится без физического копирования структур данных подготовленного оператора.

SPI_register_relation

SPI_register_relation — сделать эфемерное именованное отношение доступным по имени в запросах SPI

Синтаксис

```
int SPI_register_relation(EphemeralNamedRelation enr)
```

Описание

`SPI_register_relation` делает эфемерное именованное отношение (со связанной информацией) доступным в запросах, планируемых и выполняемых через текущее подключение SPI.

Аргументы

`EphemeralNamedRelation` *`enr`*

запись эфемерного именованного отношения в реестре

Возвращаемое значение

Если команда была выполнена успешно, возвращается следующее (неотрицательное) значение:

`SPI_OK_REL_REGISTER`

если отношение было успешно зарегистрировано по имени

В случае ошибки возвращается одно из следующих отрицательных значений:

`SPI_ERROR_ARGUMENT`

если `NULL` передан в *`enr`* или в поле `name`

`SPI_ERROR_UNCONNECTED`

если вызывается из неподключённой процедуры

`SPI_ERROR_REL_DUPLICATE`

если имя, заданное в поле `name` структуры *`enr`*, уже зарегистрировано для этого отношения

SPI_unregister_relation

SPI_unregister_relation — удалить эфемерное именованное отношение из реестра

Синтаксис

```
int SPI_unregister_relation(const char * name)
```

Описание

`SPI_unregister_relation` удаляет эфемерное именованное отношение из реестра для текущего подключения.

Аргументы

`const char *` *`name`*

имя записи отношения в реестре

Возвращаемое значение

Если команда была выполнена успешно, возвращается следующее (неотрицательное) значение:

`SPI_OK_REL_UNREGISTER`

если совокупность кортежей была успешно удалена из реестра

В случае ошибки возвращается одно из следующих отрицательных значений:

`SPI_ERROR_ARGUMENT`

если в *`name`* передан `NULL`

`SPI_ERROR_UNCONNECTED`

если вызывается из неподключённой процедуры

`SPI_ERROR_REL_NOT_FOUND`

если *`name`* не находится в реестре для текущего подключения

SPI_register_trigger_data

SPI_register_trigger_data — сделать эфемерные данные триггера доступными в запросах SPI

Синтаксис

```
int SPI_register_trigger_data(TriggerData *tdata)
```

Описание

`SPI_register_trigger_data` делает эфемерные отношения, которые перехватывает триггер, доступными для запросов, планируемых и выполняемых через текущее подключение SPI. В настоящее время это переходные таблицы, перехватываемые триггером `AFTER`, определённым с предложением `REFERENCING OLD/NEW TABLE AS`. Эта функция должна вызываться функцией, реализующей триггер на языке программирования, после подключения.

Аргументы

`TriggerData *tdata`

объект `TriggerData`, передаваемый функцией, реализующей триггер, через `fcinfo->context`

Возвращаемое значение

Если команда была выполнена успешно, возвращается следующее (неотрицательное) значение:

`SPI_OK_TD_REGISTER`

если перехваченные данные триггера (при наличии) были успешно зарегистрированы

В случае ошибки возвращается одно из следующих отрицательных значений:

`SPI_ERROR_ARGUMENT`

если в *tdata* передан `NULL`

`SPI_ERROR_UNCONNECTED`

если вызывается из неподключённой процедуры

`SPI_ERROR_REL_DUPLICATE`

если имя в любом из переходных отношений в данных триггера уже зарегистрировано для этого подключения

44.2. Вспомогательные интерфейсные функции

Функции, описанные здесь, предоставляют возможности для извлечения информации из наборов результатов, возвращаемых `SPI_execute` и другими функциями SPI.

Все функции, описанные в этом разделе, могут использоваться и в подключённых, и в неподключённых процедурах.

SPI_fname

SPI_fname — определить имя столбца с заданным номером

Синтаксис

```
char * SPI_fname(TupleDesc rowdesc, int colnumber)
```

Описание

`SPI_fname` возвращает копию имени столбца с заданным номером. (Когда эта копия имени будет не нужна, её можно освободить с помощью `pfree`.)

Аргументы

`TupleDesc` *`rowdesc`*

описание строк

`int` *`colnumber`*

номер столбца (начиная с 1)

Возвращаемое значение

Имя столбца; `NULL`, если *`colnumber`* вне допустимого диапазона. В случае ошибки в `SPI_result` устанавливается `SPI_ERROR_NOATTRIBUTE`.

SPI_fnumber

SPI_fnumber — определить номер столбца с заданным именем

Синтаксис

```
int SPI_fnumber(TupleDesc rowdesc, const char * colname)
```

Описание

`SPI_fnumber` возвращает номер столбца, имеющего заданное имя.

Если *`colname`* ссылается на системный столбец (например, `oid`), возвращается соответствующий отрицательный номер столбца. Вызывающий должен проверять, не была ли возвращена ошибка, сравнивая значение результата именно с `SPI_ERROR_NOATTRIBUTE`; проверка результата по условию меньше или равно нулю не будет корректной, если только системные столбцы не должны исключаться.

Аргументы

`TupleDesc` *`rowdesc`*

описание строк

`const char *` *`colname`*

имя столбца

Возвращаемое значение

Номер столбца (начиная с 1 для столбцов, создаваемых пользователем), либо `SPI_ERROR_NOATTRIBUTE`, если столбец с заданным именем не найден.

SPI_getvalue

SPI_getvalue — получить строковое значение указанного столбца

Синтаксис

```
char * SPI_getvalue(HeapTuple row, TupleDesc rowdesc, int colnumber)
```

Описание

`SPI_getvalue` возвращает строковое представление значения указанного столбца.

Результат возвращается в памяти, размещённой функцией `palloc`. (Когда он будет не нужен, эту память можно освободить с помощью `pfree`.)

Аргументы

`HeapTuple` *`row`*

строка с нужными данными

`TupleDesc` *`rowdesc`*

описание строк

`int` *`colnumber`*

номер столбца (начиная с 1)

Возвращаемое значение

Значение столбца, либо `NULL`, если столбец содержит NULL, *`colnumber`* вне допустимого диапазона (в `SPI_result` при этом устанавливается `SPI_ERROR_NOATTRIBUTE`) или если отсутствует функция вывода (в `SPI_result` устанавливается `SPI_ERROR_NOOUTFUNC`).

SPI_getbinval

SPI_getbinval — получить двоичное значение указанного столбца

Синтаксис

```
Datum SPI_getbinval(HeapTuple row, TupleDesc rowdesc, int colnumber,
                    bool * isnull)
```

Описание

`SPI_getbinval` возвращает значение указанного столбца во внутренней форме (в структуре `Datum`).

Эта функция не выделяет новый блок памяти для данных. В случае с типом, передаваемым по ссылке, возвращаемым значением будет указатель на переданную строку данных.

Аргументы

`HeapTuple` *`row`*

строка с нужными данными

`TupleDesc` *`rowdesc`*

описание строк

`int` *`colnumber`*

номер столбца (начиная с 1)

`bool *` *`isnull`*

признак того, что столбец содержит NULL

Возвращаемое значение

Возвращается двоичное значение столбца. Если этот столбец содержит NULL, переменной, на которую указывает *`isnull`*, присваивается true; в противном случае — false.

При ошибке в `SPI_result` устанавливается `SPI_ERROR_NOATTRIBUTE`.

SPI_gettype

SPI_gettype — получить имя типа данных указанного столбца

Синтаксис

```
char * SPI_gettype(TupleDesc rowdesc, int colnumber)
```

Описание

`SPI_gettype` возвращает копию имени типа данных указанного столбца. (Когда эта копия имени будет не нужна, её можно освободить с помощью `pfree`.)

Аргументы

`TupleDesc` *`rowdesc`*

описание строк

`int` *`colnumber`*

номер столбца (начиная с 1)

Возвращаемое значение

Имя типа данных указанного столбца, либо `NULL` в случае ошибки. При ошибке в `SPI_result` устанавливается `SPI_ERROR_NOATTRIBUTE`.

SPI_gettypeid

SPI_gettypeid — получить OID типа данных указанного столбца

Синтаксис

```
Oid SPI_gettypeid(TupleDesc rowdesc, int colnumber)
```

Описание

`SPI_gettypeid` возвращает OID типа данных указанного столбца.

Аргументы

`TupleDesc` *`rowdesc`*

описание строк

`int` *`colnumber`*

номер столбца (начиная с 1)

Возвращаемое значение

OID типа данных указанного столбца, либо `InvalidOid` в случае ошибки. При ошибке в `SPI_result` устанавливается `SPI_ERROR_NOATTRIBUTE`.

SPI_getrelname

SPI_getrelname — возвращает имя указанного отношения

Синтаксис

```
char * SPI_getrelname(Relation rel)
```

Описание

`SPI_getrelname` возвращает копию имени указанного отношения. (Когда эта копия имени будет не нужна, её можно освободить с помощью `pfree`.)

Аргументы

`Relation rel`

целевое отношение

Возвращаемое значение

Имя указанного отношения.

SPI_getnspname

SPI_getnspname — возвращает пространство имён указанного отношения

Синтаксис

```
char * SPI_getnspname(Relation rel)
```

Описание

`SPI_getnspname` возвращает копию имени пространства имён, к которому принадлежит указанное отношение (`Relation`). Пространство имён по-другому называется схемой отношения. Когда значение, возвращённое этой функцией, будет не нужно, освободите его с помощью `pfree`.

Аргументы

`Relation rel`

целевое отношение

Возвращаемое значение

Имя пространства имён указанного отношения.

44.3. Управление памятью

Postgres Pro выделяет память в *контекстах памяти* и тем самым реализует удобный способ управления выделением памяти в различных местах, с разными сроками жизни выделенной памяти. При уничтожении контекста освобождается вся выделенная в нём память. Таким образом, нет необходимости контролировать каждый отдельный объект во избежание утечек памяти; вместо этого достаточно управлять только небольшим числом контекстов. Функция `palloc` и родственные ей освобождают память из «текущего» контекста.

`SPI_connect` создаёт новый контекст памяти и делает его текущим. `SPI_finish` восстанавливает контекст, который был текущим до этого, и уничтожает контекст, созданный функцией `SPI_connect`. Эти действия обеспечивают при выходе из вашей процедуры освобождение временной памяти, выделенной внутри процедуры, во избежание утечки памяти.

Однако если ваша процедура должна вернуть объект в выделенной памяти (как значение типа, передаваемого по ссылке), эту память нельзя выделять через `palloc`, как минимум пока установлено подключение к SPI. Если вы попытаетесь это сделать, объект будет освобождён при вызове `SPI_finish` и ваша процедура не будет работать надёжно. Для решения этой проблемы выделяйте память для возвращаемого объекта, используя `SPI_palloc`. `SPI_palloc` выделяет память в «верхнем контексте исполнителя», то есть, в контексте памяти, который был текущим при вызове `SPI_connect`; именно этот контекст подходит для значения, возвращаемого из процедуры. Некоторые из вспомогательных процедур, описанных в этом разделе, также возвращают объекты, созданные в верхнем контексте исполнителя.

Когда вызывается `SPI_connect`, текущим контекстом становится частный контекст процедуры, создаваемый в `SPI_connect`. Все операции выделения памяти, выполняемые функциями `palloc`, `repalloc` или служебными функциями SPI (кроме описанных в этом разделе исключений), производятся в этом контексте. Когда процедура отключается от менеджера SPI (выполняя `SPI_finish`), текущим контекстом снова становится верхний контекст исполнителя, а вся память, выделенная в контексте процедуры, освобождается, так что использовать её дальше нельзя.

SPI_palloc

SPI_palloc — выделить память в верхнем контексте исполнителя

Синтаксис

```
void * SPI_palloc(Size size)
```

Описание

`SPI_palloc` выделяет память в верхнем контексте исполнителя.

Эту функцию можно использовать только когда установлено подключение к SPI. В противном случае она выдаёт ошибку.

Аргументы

`Size` *`size`*

размер выделяемой памяти, в байтах

Возвращаемое значение

указатель на выделенный блок памяти заданного размера

SPI_repalloc

SPI_repalloc — поменять блок памяти в верхнем контексте исполнителя

Синтаксис

```
void * SPI_repalloc(void * pointer, Size size)
```

Описание

`SPI_repalloc` изменяет размер блока памяти, ранее выделенного функцией `SPI_palloc`.

Эта функция теперь не отличается от простой `repalloc`. Она сохранена только для обратной совместимости с существующим кодом.

Аргументы

`void * pointer`

указатель на существующий блок памяти, подлежащий изменению

`Size size`

размер выделяемой памяти, в байтах

Возвращаемое значение

указатель на новый блок памяти указанного размера, в который скопировано содержимое прежнего блока

SPI_pfree

SPI_pfree — освободить память в верхнем контексте исполнителя

Синтаксис

```
void SPI_pfree(void * pointer)
```

Описание

`SPI_pfree` освобождает память, ранее выделенную функцией `SPI_palloc` или `SPI_repalloc`.

Эта функция теперь не отличается от простой `pfree`. Она сохранена только для обратной совместимости с существующим кодом.

Аргументы

`void * pointer`

указатель на существующий блок памяти, подлежащий освобождению

SPI_copytuple

SPI_copytuple — скопировать строку в верхнем контексте исполнителя

Синтаксис

```
HeapTuple SPI_copytuple(HeapTuple row)
```

Описание

`SPI_copytuple` делает копию строки в верхнем контексте исполнителя. Обычно это применяется, когда нужно вернуть изменённую строку из триггера. В функции, которая должна возвращать составной тип, нужно использовать `SPI_returntuple`.

Эту функцию можно использовать только когда установлено подключение к SPI. В противном случае она возвращает NULL и устанавливает в `SPI_result` значение `SPI_ERROR_UNCONNECTED`.

Аргументы

`HeapTuple` *`row`*

строка, подлежащая копированию

Возвращаемое значение

скопированная строка либо `NULL` в случае ошибки (`SPI_result` содержит код ошибки)

SPI_returntuple

SPI_returntuple — подготовить строку для возврата в виде Datum

Синтаксис

```
HeapTupleHeader SPI_returntuple(HeapTuple row, TupleDesc rowdesc)
```

Описание

`SPI_returntuple` делает копию строки в верхнем контексте исполнителя и возвращает её в форме типа `Datum`. Чтобы выдать результат, полученный указатель остаётся только преобразовать в `Datum` функцией `PointerGetDatum`.

Эту функцию можно использовать только когда установлено подключение к SPI. В противном случае она возвращает NULL и устанавливает в `SPI_result` значение `SPI_ERROR_UNCONNECTED`.

Заметьте, что эту операцию следует применять в функциях, объявленных как возвращающие составные типы. В триггерах она не применяется; чтобы вернуть изменённую строку из триггера, используйте `SPI_copytuple`.

Аргументы

`HeapTuple` *`row`*

строка, подлежащая копированию

`TupleDesc` *`rowdesc`*

дескриптор строки (передавайте каждый раз один дескриптор для более эффективного кеширования)

Возвращаемое значение

`HeapTupleHeader`, указывающий на скопированную строку, или `NULL` в случае ошибки (`SPI_result` содержит код ошибки)

SPI_modifytuple

SPI_modifytuple — создать строку, заменяя отдельные поля в данной

Синтаксис

```
HeapTuple SPI_modifytuple(Relation rel, HeapTuple row, int ncols,
                          int * colnum, Datum * values, const char * nulls)
```

Описание

`SPI_modifytuple` создаёт новую строку, подставляя новые значения для указанных столбцов и копируя исходное содержимое остальных столбцов. Исходная строка не изменяется. Новая строка возвращается в верхнем контексте исполнителя.

Эту функцию можно использовать только когда установлено подключение к SPI. В противном случае она возвращает NULL и устанавливает в `SPI_result` значение `SPI_ERROR_UNCONNECTED`.

Аргументы

`Relation` *`rel`*

Используется только в качестве дескриптора строки. (Передача отношения вместо собственно дескриптора строки — нехорошая особенность.)

`HeapTuple` *`row`*

строка, подлежащая изменению

`int` *`ncols`*

число изменяемых столбцов

`int *` *`colnum`*

массив длины *`ncols`*, содержащий номера изменяемых столбцов (начиная с 1)

`Datum *` *`values`*

массив длины *`ncols`*, содержащий новые значения указанных столбцов

`const char *` *`nulls`*

массив длины *`ncols`*, описывающий, в каких столбцах передаётся NULL

Если в *`nulls`* передаётся `NULL`, `SPI_modifytuple` считает, что ни один из параметров не равен NULL. В противном случае элемент массива *`nulls`* должен содержать `' '`, если значение соответствующего параметра не NULL, либо `'n'`, если это значение — NULL. (В последнем случае значение, переданное в соответствующем элементе *`values`*, не учитывается.) Заметьте, что *`nulls`* — это не текстовая строка, а просто массив: ноль `'\0'` в конце не нужен.

Возвращаемое значение

новая строка с изменениями, размещённая в верхнем контексте исполнителя, или `NULL` при ошибке (`SPI_result` содержит код ошибки)

В случае ошибки в `SPI_result` устанавливается:

`SPI_ERROR_ARGUMENT`

если *`rel`* — `NULL`, либо *`row`* — `NULL`, либо *`ncols`* меньше или равно 0, либо *`colnum`* — `NULL`, либо *`values`* — `NULL`

`SPI_ERROR_NOATTRIBUTE`

если *`colnum`* содержит недопустимый номер столбца (меньше или равен 0, либо больше числа столбцов в строке *`row`*)

`SPI_ERROR_UNCONNECTED`

если SPI неактивен

SPI_freetuple

SPI_freetuple — освободить строку, размещённую в верхнем контексте исполнителя

Синтаксис

```
void SPI_freetuple(HeapTuple row)
```

Описание

`SPI_freetuple` освобождает строку, ранее размещённую в верхнем контексте исполнителя.

Эта функция теперь не отличается от простой `heap_freetuple`. Она сохранена только для обратной совместимости с существующим кодом.

Аргументы

`HeapTuple` *`row`*

строка, подлежащая освобождению

SPI_freetuptable

SPI_freetuptable — освободить набор строк, созданный `SPI_execute` или подобной функцией

Синтаксис

```
void SPI_freetuptable(SPITupleTable * tuptable)
```

Описание

`SPI_freetuptable` освобождает набор строк, созданных предыдущей функцией SPI выполнения команд, например `SPI_execute`. Таким образом, при вызове этой функции в качестве аргумента часто передаётся глобальная переменная `SPI_tuptable`.

Эта функция полезна, когда процедура, использующая SPI, должна выполнить несколько команд, но не хочет сохранять результаты предыдущих команд до завершения. Заметьте, что любые не освобождённые таким образом наборы строк будут всё равно освобождены при выполнении `SPI_finish`. Кроме того, если была запущена подтранзакция, а затем она прервалась в ходе выполнения процедуры SPI, все наборы строк, созданные в рамках подтранзакции, будут автоматически освобождены.

Начиная с PostgreSQL версии 9.3, `SPI_freetuptable` содержит защитную логику, отфильтровывающую повторные запросы на удаление одного и того же набора строк. В предыдущих версиях повторное удаление могло приводить к сбоям.

Аргументы

`SPITupleTable * tuptable`

указатель на набор строк, который нужно освободить (если NULL, ничего не происходит)

SPI_freeplan

SPI_freeplan — освободить ранее сохранённый подготовленный оператор

Синтаксис

```
int SPI_freeplan(SPIPlanPtr plan)
```

Описание

`SPI_freeplan` освобождает подготовленный оператор, до этого выданный функцией `SPI_prepare` или сохранённый функциями `SPI_keepplan` и `SPI_saveplan`.

Аргументы

`SPIPlanPtr` *`plan`*

указатель на оператор, подлежащий освобождению

Возвращаемое значение

0 в случае успеха; `SPI_ERROR_ARGUMENT`, если *`plan`* неверный или `NULL`

44.4. Видимость изменений в данных

Видимость изменений в данных, которые производятся функциями, использующими SPI, (или любыми другими функциями на C), описывается следующими правилами:

- В процессе выполнения SQL-команды любые произведённые ей изменения не видны для неё самой. Например, в команде:

  ```
  INSERT INTO a SELECT * FROM a;
  ```

  вставляемые строки не видны в части `SELECT`.
- Изменения, произведённые командой K, видны во всех командах, запущенных после K, независимо от того, были ли эти команды запущены из K (во время выполнения K) или после завершения K.
- Команды, выполняемые через SPI внутри функции, вызванной SQL-командой (будь то обычная функция или триггер), следуют одному или другому из вышеприведённых правил в зависимости флага чтения/записи, переданного SPI. Команды, выполняемые в режиме «только чтение», следует первому правилу: они не видят изменений, произведённых вызывающей командой. Команды, выполняемые в режиме «чтение-запись», следуют второму правилу: они могут видеть все произведённые к этому времени изменения.
- Все стандартные процедурные языки устанавливают режим чтения-записи в SPI в зависимости от атрибута изменчивости функции. Команды функций `STABLE` и `IMMUTABLE` выполняются в режиме «только чтение», тогда как команды функций `VOLATILE` — в режиме «чтение-запись». Хотя авторы функций на C могут нарушить это соглашение, вряд ли это будет хорошей идеей.

В следующем разделе приводится пример, иллюстрирующий применение этих правил.

44.5. Примеры

Этот раздел содержит очень простой пример использования SPI. Процедура `execq` принимает в качестве первого аргумента команду SQL, а в качестве второго число строк, выполняет команду, вызывая `SPI_exec`, и возвращает число строк, обработанных этой командой. Более сложные примеры работы с SPI вы можете найти в `src/test/regress/regress.c` в дереве исходного кода, а также в модуле spi.

```
#include "postgres.h"

#include "executor/spi.h"
```

```
#include "utils/builtins.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(execq);

Datum
execq(PG_FUNCTION_ARGS)
{
    char *command;
    int cnt;
    int ret;
    uint64 proc;

    /* Преобразовать данный текстовый объект в строку C */
    command = text_to_cstring(PG_GETARG_TEXT_PP(0));
    cnt = PG_GETARG_INT32(1);

    SPI_connect();

    ret = SPI_exec(command, cnt);

    proc = SPI_processed;

    /*
     * Если были выбраны какие-то строки, вывести их через elog(INFO).
     */
    if (ret > 0 && SPI_tuptable != NULL)
    {
        TupleDesc tupdesc = SPI_tuptable->tupdesc;
        SPITupleTable *tuptable = SPI_tuptable;
        char buf[8192];
        uint64 j;

        for (j = 0; j < proc; j++)
        {
            HeapTuple tuple = tuptable->vals[j];
            int i;

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                snprintf(buf + strlen(buf), sizeof(buf) - strlen(buf), " %s%s",
                        SPI_getvalue(tuple, tupdesc, i),
                        (i == tupdesc->natts) ? " " : " |");
            elog(INFO, "EXECQ: %s", buf);
        }
    }

    SPI_finish();
    pfree(command);

    PG_RETURN_INT64(proc);
}
```

Так эта функция будет объявляться после того, как она будет скомпилирована в разделяемую библиотеку (подробности в Подразделе 35.9.5):

```
CREATE FUNCTION execq(text, integer) RETURNS int8
```

```
    AS 'имя_файла'
    LANGUAGE C STRICT;
```

Демонстрация использования:

```
=> SELECT execq('CREATE TABLE a (x integer)', 0);
 execq
-------
     0
(1 row)

=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
INSERT 0 1
=> SELECT execq('SELECT * FROM a', 0);
INFO:  EXECQ:  0    -- вставлено функцией execq
INFO:  EXECQ:  1    -- возвращено функцией execq и вставлено командой INSERT

 execq
-------
     2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a', 1);
 execq
-------
     1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO:  EXECQ:  0
INFO:  EXECQ:  1
INFO:  EXECQ:  2    -- 0 + 2, вставлена только одна строка - как указано

 execq
-------
     3              -- 10 — только максимальное значение, 3 — реальное число строк
(1 row)

=> DELETE FROM a;
DELETE 3
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1                  -- нет строк в a (0) + 1
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO:  EXECQ:  1
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1
 2                  -- была одна строка в a + 1
(2 rows)

-- Этот пример демонстрирует правило видимости изменений в данных:
```

```
=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  2
INSERT 0 2
=> SELECT * FROM a;
 x
---
 1
 2
 2                  -- 2 строки * 1 (x в первой в строке)
 6                  -- 3 строки (2 + 1 только вставленная) * 2 (x во второй строке)
(4 rows)               ^^^^^^
                       строки, видимые в execq() при разных вызовах
```

Глава 45. Фоновые рабочие процессы

Postgres Pro поддерживает расширенную возможность запускать пользовательский код в отдельных процессах. Такие процессы запускаются, останавливаются и контролируются главным процессом `postgres`, который позволяет тесно связать их жизненный цикл с состоянием сервера. Эти процессы могут получать доступ к области разделяемой памяти Postgres Pro и устанавливать внутренние подключения к базам данных; они также могут последовательно запускать транзакции, как и обычные серверные процессы, обслуживающие клиентов. Кроме того, используя libpq, они могут подключаться к серверу и работать как обычные клиентские приложения.

> **Предупреждение**
>
> С использованием фоновых рабочих процессов сопряжены угрозы стабильности и безопасности, так как они реализуются на языке `C`, и значит имеют неограниченный доступ к данным. Администраторы, желающие использовать модули, в которых задействованы фоновые рабочие процессы, должны быть крайне осторожными. Запускать рабочие процессы можно разрешать только модулям, прошедшим всесторонний аудит.

Рабочие процессы могут инициализироваться во время запуска Postgres Pro, если имя соответствующего модуля добавлено в `shared_preload_libraries`. Модуль, желающий запустить фоновый процесс, может зарегистрировать его, вызвав `RegisterBackgroundWorker(BackgroundWorker *worker)` из своей функции `_PG_init()`. Рабочие процессы также могут быть запущены после запуска системы с помощью функции `RegisterDynamicBackgroundWorker(BackgroundWorker *worker, BackgroundWorkerHandle **handle)`. В отличие от функции `RegisterBackgroundWorker`, которую можно вызывать только из управляющего процесса, `RegisterDynamicBackgroundWorker` должна вызываться из обычного обслуживающего процесса.

Структура `BackgroundWorker` определяется так:

```
typedef void (*bgworker_main_type)(Datum main_arg);
typedef struct BackgroundWorker
{
    char        bgw_name[BGW_MAXLEN];
    int         bgw_flags;
    BgWorkerStartTime bgw_start_time;
    int         bgw_restart_time;   /* время в секундах либо BGW_NEVER_RESTART */
    char        bgw_library_name[BGW_MAXLEN];
    char        bgw_function_name[BGW_MAXLEN];
    Datum       bgw_main_arg;
    char        bgw_extra[BGW_EXTRALEN];
    int         bgw_notify_pid;
} BackgroundWorker;
```

Поле `bgw_name` содержит строку, выводимую в отладочных сообщениях, списках процессов и подобных контекстах.

Поле `bgw_flags` представляет битовую маску, обозначающую запрашиваемые модулем возможности. Допустимые в нём флаги:

`BGWORKER_SHMEM_ACCESS`

Запрашивается доступ к общей памяти. Рабочие процессы без доступа к общей памяти не могут обращаться к общим структурам данных Postgres Pro, в частности, к обычным и лёгким блокировкам, общим буферам, или каким-либо структурам данным, которые рабочий процесс может создавать для собственного пользования.

BGWORKER_BACKEND_DATABASE_CONNECTION

Запрашивается возможность устанавливать подключение к базе данных, через которое можно запускать транзакции и запросы. Рабочий процесс, использующий BGWORKER_BACKEND_DATABASE_CONNECTION для подключения к базе данных, должен также запросить доступ к разделяемой памяти, установив BGWORKER_SHMEM_ACCESS; в противном случае процесс не запустится.

В bgw_start_time определяется состояние сервера, в котором postgres должен запустить этот процесс; возможные варианты: BgWorkerStart_PostmasterStart (выполнить запуск сразу после того, как postgres завершит инициализацию; процессы, выбирающие такой режим, не могут подключаться к базам данных), BgWorkerStart_ConsistentState (выполнить запуск, когда будет достигнуто согласованное состояние горячего резерва, и когда процессы могут подключаться к базам данных и выполнять запросы на чтение), и BgWorkerStart_RecoveryFinished (выполнить запуск, как только система перейдёт в обычный режим чтения-записи). Заметьте, что два последних варианта различаются только для серверов горячего резерва. Заметьте также, что этот параметр указывает только, когда должны запускаться процессы; при переходе в другое состояние они не будут останавливаться.

bgw_restart_time задаёт паузу (в секундах), которую должен сделать postgres, прежде чем перезапускать процесс в случае его отказа. Это может быть любое положительное значение, либо BGW_NEVER_RESTART, указывающее, что процесс не нужно перезапускать в случае сбоя.

bgw_library_name определяет имя библиотеки, в которой следует искать точку входа для запуска рабочего процесса. Указанная библиотека будет динамически загружена рабочим процессом, а вызываемая функция будет выбрана по имени bgw_function_name. Для функции, загружаемой из кода ядра, в этом поле должно быть «postgres».

bgw_function_name определяет имя функции в динамически загружаемой библиотеке, которая будет точкой входа в новый рабочий процесс.

В bgw_main_arg задаётся аргумент Datum, передаваемый основной функции фонового процесса. Эта функция должна принимать один аргумент типа Datum и возвращать void. В качестве этого аргумента ей и передаётся bgw_main_arg. Кроме того, глобальная переменная MyBgworkerEntry указывает на копию структуры BackgroundWorker, переданной при регистрации; содержимое этой структуры может быть полезно рабочему процессу.

В Windows (и везде, где определяется EXEC_BACKEND) или в динамических рабочих процессах передавать Datum по ссылке небезопасно, возможна только передача по значению. Поэтому если функции требуется аргумент, наиболее безопасно будет передать int32 или другое небольшое значение, содержащее индекс в массиве, размещённом в разделяемой памяти. Если же попытаться передать значение cstring или text, этот указатель нельзя будет использовать в новом рабочем процессе.

Поле bgw_extra может содержать дополнительные данные, передаваемые фоновому рабочему процессу. В отличие от bgw_main_arg, эти данные не передаются в качестве аргумента основной функции рабочего процесса, но могут быть получены через MyBgworkerEntry, как описывалось выше.

В bgw_notify_pid задаётся PID обслуживающего процесса Postgres Pro, которому главный процесс должен посылать сигнал SIGUSR1 при запуске и завершении нового рабочего процесса. Это поле должно содержать 0 для рабочих процессов, регистрируемых при запуске главного процесса, либо когда обслуживающий процесс не желает ждать окончания запуска рабочего процесса. Во всех остальных случаях в нём должно быть значение MyProcPid.

Запущенный процесс может подключиться к базе данных, вызвав BackgroundWorkerInitializeConnection(*char *dbname*, *char *username*) или BackgroundWorkerInitializeConnectionByOid(*Oid dboid*, *Oid useroid*). Через это подключение процесс сможет выполнять транзакции и запросы, используя функции SPI. Если в dbname

передаётся NULL или dboid равен InvalidOid, сеанс не подключается ни к какой конкретной базе данных, но может обращаться к общим каталогам. Если в username передаётся NULL или useroid равен InvalidOid, процесс будет действовать от имени суперпользователя, созданного во время initdb. Рабочий процесс может вызывать только одну из двух этих функций и только один раз. Переключаться между базами данных он не может.

Сигналы изначально блокируются при вызове основной функции рабочего процесса и должны быть разблокированы ей: это позволяет процессу при необходимости настроить собственные обработчики событий. Новый процесс может разблокировать сигналы, вызвав BackgroundWorkerUnblockSignals, и заблокировать их, вызвав BackgroundWorkerBlockSignals.

Если bgw_restart_time для рабочего процесса имеет значение BGW_NEVER_RESTART, либо он завершается с кодом выхода 0, либо если его работа заканчивается вызовом TerminateBackgroundWorker, он автоматически перестаёт контролироваться управляющим процессом при выходе. В противном случае он будет перезапущен через время, заданное в bgw_restart_time, либо немедленно, если управляющему серверу пришлось переинициализировать кластер из-за сбоя обслуживающего процесса. Обслуживающие процессы, которым нужно только приостановить своё выполнение на время, должны переходить в состояние прерываемого ожидания, а не завершаться; для этого используется функция WaitLatch(). При вызове этой функции обязательно установите флаг WL_POSTMASTER_DEATH и проверьте код возврата, чтобы корректно выйти в экстренном случае, когда был завершён сам postgres.

Когда рабочий процесс регистрируется функцией RegisterDynamicBackgroundWorker, обслуживающий процесс, производящий эту регистрацию, может получить информацию о состоянии порождённого процесса. Обслуживающие процессы, желающие сделать это, должны передать адрес BackgroundWorkerHandle * во втором аргументе RegisterDynamicBackgroundWorker. Если рабочий процесс успешно зарегистрирован, по этому адресу будет записан указатель на скрытую структуру, который можно затем передать функции GetBackgroundWorkerPid(*BackgroundWorkerHandle* *, *pid_t* *) или TerminateBackgroundWorker(*BackgroundWorkerHandle* *). Вызывая GetBackgroundWorkerPid, можно опрашивать состояние рабочего процесса: значение результата BGWH_NOT_YET_STARTED показывает, что рабочий процесс ещё не запущен управляющим; BGWH_STOPPED показывает, что он был запущен, но сейчас не работает; и BGWH_STARTED показывает, что он работает в данный момент. В последнем случае через второй аргумент также возвращается PID этого процесса. Обрабатывая вызов TerminateBackgroundWorker, управляющий процесс посылает SIGTERM рабочему процессу, если он работает, и перестаёт его контролировать сразу по его завершении.

В некоторых случаях процессу, регистрирующему рабочий процесс, может потребоваться дождаться завершения запуска этого процесса. Это можно реализовать, записав в bgw_notify_pid значение MyProcPid, а затем передав указатель BackgroundWorkerHandle *, полученный во время регистрации, функции WaitForBackgroundWorkerStartup(*BackgroundWorkerHandle* **handle*, *pid_t* *). Эта функция заблокирует выполнение, пока управляющий процесс не попытается запустить рабочий процесс, либо пока сам управляющий процесс не завершится. Если рабочий процесс запущен, возвращается значение BGWH_STARTED, и по переданному адресу записывается его PID. В противном случае возвращается значение BGWH_STOPPED или BGWH_POSTMASTER_DIED.

Если фоновый рабочий процесс передаёт асинхронные уведомления, вызывая команду NOTIFY через SPI (Server Programming Interface, Интерфейс программирования сервера), он должен явно вызвать ProcessCompletedNotifies после фиксации окружающей транзакции, чтобы все эти уведомления были доставлены. Если рабочий процесс зарегистрируется для получения асинхронных уведомлений, вызвав LISTEN через SPI, уведомления будут выводиться, но перехватить и обработать эти уведомления программным образом нет возможности.

Рабочий пример, демонстрирующий некоторые полезные приёмы, можно найти в модуле src/test/modules/worker_spi.

Максимальное число рабочих процессов, которые можно зарегистрировать, ограничивается значением max_worker_processes.

Глава 46. Логическое декодирование

Postgres Pro обеспечивает инфраструктуру для потоковой передачи изменений, выполняемых через SQL, внешним потребителям. Эта функциональность может быть полезна для самых разных целей, включая аудит и реализацию репликации.

Изменения передаются в потоках, связываемых со слотами логической репликации.

Формат, в котором передаются изменения, определяет используемый модуль вывода. Пример модуля вывода включён в дистрибутив Postgres Pro. Также возможно разработать и другие модули, расширяющие выбор доступных форматов, не затрагивая код ядра самого сервера. Любой модуль вывода получает на вход отдельные строки, создаваемые командой `INSERT`, и новые версии строк, которые создаёт `UPDATE`. Доступность старых версий строк для `UPDATE` и `DELETE` зависит от выбора варианта идентификации реплики (см. описание `REPLICA IDENTITY`).

Изменения могут быть получены либо по протоколу потоковой репликации (см. Раздел 50.4 и Раздел 46.3), либо через функции, вызываемые в SQL (см. Раздел 46.4). Также возможно разработать дополнительные методы для обработки данных, поступающих через слот репликации, не модифицируя код ядра сервера (см. Раздел 46.7).

46.1. Примеры логического декодирования

Следующий пример демонстрирует управление логическим декодированием на уровне SQL.

Прежде чем вы сможете использовать логическое декодирование, вы должны установить в wal_level значение `logical`, а в max_replication_slots значение, не меньшее 1. После этого вы должны подключиться к целевой базе данных (в следующем примере, это `postgres`) как суперпользователь.

```
postgres=# -- Создать слот с именем 'regression_slot', использующий модуль вывода
 'test_decoding'
postgres=# SELECT * FROM pg_create_logical_replication_slot('regression_slot',
 'test_decoding');
    slot_name    |    lsn
-----------------+-----------
 regression_slot | 0/16B1970
(1 row)

postgres=# SELECT slot_name, plugin, slot_type, database, active, restart_lsn,
 confirmed_flush_lsn FROM pg_replication_slots;
    slot_name    |    plugin     | slot_type | database | active | restart_lsn |
 confirmed_flush_lsn
-----------------+---------------+-----------+----------+--------+-------------
+-----------------
 regression_slot | test_decoding | logical   | postgres | f      | 0/16A4408   |
 0/16A4440
(1 row)

postgres=# -- Пока никакие изменения не видны
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
 lsn | xid | data
-----+-----+------
(0 rows)

postgres=# CREATE TABLE data(id serial primary key, data text);
CREATE TABLE

postgres=# -- DDL не реплицируется, поэтому видна только транзакция
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
```

```
    lsn    |  xid  |     data
-----------+-------+--------------
 0/BA2DA58 | 10297 | BEGIN 10297
 0/BA5A5A0 | 10297 | COMMIT 10297
(2 rows)

postgres=# -- Когда изменения прочитаны, они считаются обработанными и уже не выдаются
postgres=# -- в последующем вызове:
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
 lsn | xid | data
-----+-----+------
(0 rows)

postgres=# BEGIN;
postgres=# INSERT INTO data(data) VALUES('1');
postgres=# INSERT INTO data(data) VALUES('2');
postgres=# COMMIT;

postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |                          data
-----------+-------+---------------------------------------------------------
 0/BA5A688 | 10298 | BEGIN 10298
 0/BA5A6F0 | 10298 | table public.data: INSERT: id[integer]:1 data[text]:'1'
 0/BA5A7F8 | 10298 | table public.data: INSERT: id[integer]:2 data[text]:'2'
 0/BA5A8A8 | 10298 | COMMIT 10298
(4 rows)

postgres=# INSERT INTO data(data) VALUES('3');

postgres=# -- Также можно заглянуть вперёд в потоке изменений, не считывая эти
 изменения
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |                          data
-----------+-------+---------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299
(3 rows)

postgres=# -- Следующий вызов pg_logical_slot_peek_changes() снова возвращает те же
 изменения
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |                          data
-----------+-------+---------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299
(3 rows)

postgres=# -- Модулю вывода можно передать параметры, влияющие на форматирование
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL, NULL,
 'include-timestamp', 'on');
    lsn    |  xid  |                          data
-----------+-------+---------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299 (at 2017-05-10 12:07:21.272494-04)
(3 rows)
```

```
postgres=# -- Не забудьте удалить слот, который вам больше не нужен, чтобы он
postgres=# -- не потреблял ресурсы сервера:
postgres=# SELECT pg_drop_replication_slot('regression_slot');
 pg_drop_replication_slot
--------------------------

(1 row)
```

Следующий пример показывает, как можно управлять логическим декодированием средствами протокола потоковой репликации, используя программу pg_recvlogical, включённую в дистрибутив Postgres Pro. Для этого нужно, чтобы конфигурация аутентификации клиентов допускала подключения для репликации (см. Подраздел 25.2.5.1) и чтобы значение `max_wal_senders` было достаточно большим и позволило установить дополнительное подключение.

```
$ pg_recvlogical -d postgres --slot test --create-slot
$ pg_recvlogical -d postgres --slot test --start -f -
Control+Z
$ psql -d postgres -c "INSERT INTO data(data) VALUES('4');"
$ fg
BEGIN 693
table public.data: INSERT: id[integer]:4 data[text]:'4'
COMMIT 693
Control+C
$ pg_recvlogical -d postgres --slot test --drop-slot
```

46.2. Концепции логического декодирования

46.2.1. Логическое декодирование

Логическое декодирование — это процедура извлечения всех постоянных изменений, происходящих в таблицах базы данных, в согласованном и понятном формате, который можно интерпретировать, не имея полного представления о внутреннем состоянии базы данных.

В Postgres Pro логическое декодирование реализуется путём перевода содержимого журнала предзаписи, описывающего изменения на уровне хранения, в специальную форму уровня приложения, например, в поток кортежей или операторов SQL.

46.2.2. Слоты репликации

В контексте логической репликации слот представляет поток изменений, которые могут быть воспроизведены клиентом в том порядке, в каком они происходили на исходном сервере. Через каждый слот передаётся последовательность изменений в одной базе данных.

> **Примечание**
>
> В Postgres Pro также есть слоты потоковой репликации (см. Подраздел 25.2.5), но они используются несколько по-другому.

Слоту репликации назначается идентификатор, уникальный для всех баз данных в кластере Postgres Pro. Слоты сохраняются независимо от подключений, использующих их, и защищены от сбоев сервера.

При обычных условиях через логический слот каждое изменение передаётся только один раз. Текущая позиция в каждом слоте сохраняется только в контрольной точке, так что в случае сбоя слот может вернуться к предыдущему LSN, вследствие чего последние изменения могут быть переданы повторно при перезапуске сервера. За исключение нежелательных эффектов от повторной обработки одного и того же сообщения отвечают клиенты логического декодирования.

Клиенты могут запоминать при декодировании, какой последний LSN они уже получали, и пропускать повторяющиеся данные или (при использовании протокола репликации) запрашивать, чтобы декодирование начиналось с этого LSN, а не с позиции, выбираемой сервером. Для этого разработан механизм отслеживания репликации, о котором можно узнать подробнее в описании источников репликации.

Для одной базы данных могут существовать несколько независимых слотов. Каждый слот имеет собственное состояние, что позволяет различным потребителям получать изменения с разных позиций в потоке изменений базы данных. Для большинства приложений каждому потребителю требуется отдельный слот.

Слот логической репликации ничего не знает о состоянии получателя(ей). Возможно даже иметь несколько различных потребителей одного слота в разные моменты времени; они просто будут получать изменения с момента, когда их перестал получать предыдущий потребитель. Но в любой определённый момент получать изменения может только один потребитель.

> **Примечание**
>
> Слоты репликации сохраняются при сбоях сервера и ничего не знают о состоянии их потребителя(ей). Они не дают удалять требуемые ресурсы, даже когда не используются никаким подключением. На это уходит место в хранилище, так как ни сегменты WAL, ни требуемые строки из системных каталогов нельзя будет удалить в результате `VACUUM`, пока они нужны этому слоту репликации. Поэтому, если слот больше не требуется, его следует ликвидировать.

46.2.3. Модули вывода

Модули вывода переводят данные из внутреннего представления в журнале предзаписи в формат, устраивающий потребителя слота репликации.

46.2.4. Экспортированные снимки

Когда новый слот репликации создаётся через интерфейс потоковой репликации, экспортируется снимок (см. CREATE_REPLICATION_SLOT), который будет показывать ровно то состояние базы данных, изменения после которого будут включаться в поток изменений. Используя его, можно создать новую реплику, воспользовавшись командой `SET TRANSACTION SNAPSHOT`, чтобы получить состояние базы в момент создания слота. После этого данную транзакцию можно использовать для выгрузки состояния базы на момент экспорта снимка, а затем изменять это состояние, применяя содержимое слота, так что никакие изменения не будут потеряны.

Создание снимка возможно не всегда. В частности, невозможно создать снимок при подключении к горячему резерву. Приложения, которым не требуется экспорт снимка, могут подавить его, воспользовавшись указанием `NOEXPORT_SNAPSHOT`.

46.3. Интерфейс протокола потоковой репликации

Команды

- `CREATE_REPLICATION_SLOT` *`имя_слота`* `LOGICAL` *`модуль_вывода`*
- `DROP_REPLICATION_SLOT` *`имя_слота`* [`WAIT`]
- `START_REPLICATION SLOT` *`имя_слота`* `LOGICAL ...`

применяются для создания, удаления и передачи изменений из слота репликации, соответственно. Эти команды доступны только для соединения репликации; их нельзя использовать в обычном SQL. Подробнее они описаны в Разделе 50.4.

Для управления логическим декодированием по соединению потоковой репликации можно применять программу pg_recvlogical. (Внутри неё используются эти команды.)

46.4. Интерфейс логического декодирования на уровне SQL

Подробнее API уровня SQL для взаимодействия с механизмом логическим декодированием описан в Подразделе 9.26.6.

Синхронная репликация (см. Подраздел 25.2.8) поддерживается только для слотов репликации, которые используются через интерфейс потоковой репликации. Интерфейс функций и дополнительные, не системные интерфейсы не поддерживают синхронную репликацию.

46.5. Системные каталоги, связанные с логическим декодированием

Информацию о текущем состоянии слотов репликации и соединений потоковой репликации отображают представления `pg_replication_slots` и `pg_stat_replication`, соответственно. Эти представления относятся и к физической, и к логической репликации.

46.6. Модули вывода логического декодирования

Пример модуля вывода можно найти в подкаталоге `contrib/test_decoding` в дереве исходного кода Postgres Pro.

46.6.1. Функция инициализации

Модуль вывода загружается в результате динамической загрузки разделяемой библиотеки (при этом в качестве имени библиотеки задаётся имя модуля). Для нахождения библиотеки применяется обычный путь поиска библиотек. В этой библиотеке должна быть функция `_PG_output_plugin_init`, которая показывает, что библиотека на самом деле представляет собой модуль вывода, и устанавливает требуемые обработчики модуля вывода. Этой функции передаётся структура, в которой должны быть заполнены указатели на функции-обработчики отдельных действий.

```
typedef struct OutputPluginCallbacks
{
    LogicalDecodeStartupCB startup_cb;
    LogicalDecodeBeginCB begin_cb;
    LogicalDecodeChangeCB change_cb;
    LogicalDecodeCommitCB commit_cb;
    LogicalDecodeMessageCB message_cb;
    LogicalDecodeFilterByOriginCB filter_by_origin_cb;
    LogicalDecodeShutdownCB shutdown_cb;
} OutputPluginCallbacks;

typedef void (*LogicalOutputPluginInit) (struct OutputPluginCallbacks *cb);
```

Обработчики `begin_cb`, `change_cb` и `commit_cb` должны устанавливаться обязательно, а `startup_cb`, `filter_by_origin_cb` и `shutdown_cb` могут отсутствовать.

46.6.2. Возможности

Для декодирования, форматирования и вывода изменений модули вывода могут использовать практически всю обычную инфраструктуру сервера, включая вызов функций вывода типов. К отношениям разрешается доступ только на чтение, если только эти отношения были созданы программой `initdb` в схеме `pg_catalog`, либо помечены как пользовательские таблицы каталогов командами

```
ALTER TABLE user_catalog_table SET (user_catalog_table = true);
CREATE TABLE another_catalog_table(data text) WITH (user_catalog_table = true);
```

Любые действия, которые требуют присваивания идентификатора транзакции, запрещаются. В частности, к этим действиям относятся операции записи в таблицы, изменения DDL и вызов `txid_current()`.

46.6.3. Режимы вывода

Обработчики в модуле вывода могут передавать данные потребителю в практически любых форматах. Для некоторых вариантов использования, например, просмотра изменений через SQL, вывод информации в типах, которые могут содержать произвольные данные (например, `bytea`), может быть неудобоваримым. Если модуль вывода выводит только текстовые данные в кодировке сервера, он может объявить это, установив в `OutputPluginOptions.output_type` значение `OUTPUT_PLUGIN_TEXTUAL_OUTPUT` вместо `OUTPUT_PLUGIN_BINARY_OUTPUT` в обработчике запуска. В этом случае все данные должны быть в кодировке сервера, чтобы их можно было передать в значении типа `text`. Это контролируется в сборках с включёнными проверочными утверждениями.

46.6.4. Обработчики в модуле вывода

Модуль вывода уведомляется о происходящих изменениях через различные обработчики, которые он должен установить.

Параллельные транзакции декодируются в порядке фиксирования, при этом только изменения, относящиеся к определённой транзакции, декодируются между вызовами обработчиков `begin` и `commit`. Транзакции, отменённые явно или неявно, никогда не декодируются. Успешные точки сохранения заворачиваются в транзакцию, содержащую их, в том порядке, в како они выполнялись в этой транзакции.

> **Примечание**
>
> Декодироваться будут только те транзакции, которые уже успешно сброшены на диск. Вследствие этого, `COMMIT` может не декодироваться в следующем сразу за ним вызове `pg_logical_slot_get_changes()`, когда `synchronous_commit` имеет значение `off`.

46.6.4.1. Обработчик запуска

Необязательный обработчик `startup_cb` вызывается, когда слот репликации создаётся или через него запрашивается передача изменений, независимо от того, в каком количестве изменения готовы к передаче.

```
typedef void (*LogicalDecodeStartupCB) (struct LogicalDecodingContext *ctx,
                                        OutputPluginOptions *options,
                                        bool is_init);
```

Параметр `is_init` будет равен true, когда слот репликации создаётся, и false в противном случае. Параметр *`options`* указывает на структуру параметров, которые могут устанавливать модули вывода:

```
typedef struct OutputPluginOptions
{
    OutputPluginOutputType output_type;
} OutputPluginOptions;
```

В поле `output_type` должно быть значение `OUTPUT_PLUGIN_TEXTUAL_OUTPUT` или `OUTPUT_PLUGIN_BINARY_OUTPUT`. См. также Подраздел 46.6.3.

Обработчик запуска должен проверить параметры, представленные в `ctx->output_plugin_options`. Если модулю вывода требуется поддерживать состояние, он может сохранить его в `ctx->output_plugin_private`.

46.6.4.2. Обработчик выключения

Необязательный обработчик `shutdown_cb` вызывается, когда ранее активный слот репликации перестаёт использоваться, так что ресурсы, занятые модулем вывода, можно освободить. При этом слот не обязательно удаляется, прекращается только потоковая передача через него.

```
typedef void (*LogicalDecodeShutdownCB) (struct LogicalDecodingContext *ctx);
```

46.6.4.3. Обработчик начала транзакции

Обязательный обработчик `begin_cb` вызывается, когда декодируется начало зафиксированной транзакции. Прерванные транзакции и их содержимое никогда не декодируется.

```
typedef void (*LogicalDecodeBeginCB) (struct LogicalDecodingContext *ctx,
                                      ReorderBufferTXN *txn);
```

Параметр *`txn`* содержит метаинформацию о транзакции, в частности её идентификатор и время её фиксирования.

46.6.4.4. Обработчик завершения транзакции

Обязательный обработчик `commit_cb` вызывается, когда декодируется фиксирование транзакции. Перед этим обработчиком будет вызываться обработчик `change_cb` для всех изменённых строк (если строки были изменены).

```
typedef void (*LogicalDecodeCommitCB) (struct LogicalDecodingContext *ctx,
                                       ReorderBufferTXN *txn,
                                       XLogRecPtr commit_lsn);
```

46.6.4.5. Обработчик изменения

Обязательный обработчик `change_cb` вызывается для каждого отдельного изменения строки в транзакции, производимого командами `INSERT`, `UPDATE` или `DELETE`. Даже если команда изменила несколько строк сразу, этот обработчик будет вызываться для каждой отдельной строки.

```
typedef void (*LogicalDecodeChangeCB) (struct LogicalDecodingContext *ctx,
                                       ReorderBufferTXN *txn,
                                       Relation relation,
                                       ReorderBufferChange *change);
```

Параметры *`ctx`* и *`txn`* имеют то же содержимое, что и для обработчиков `begin_cb` и `commit_cb`; дополнительный дескриптор отношения *`relation`* указывает на отношение, к которому принадлежит строка, а структура *`change`* описывает передаваемое изменение строки.

> **Примечание**
>
> В процессе логического декодирования могут быть обработаны изменения только в таблицах, не являющихся нежурналируемыми (см. описание `UNLOGGED`) или временными (см. описание `TEMPORARY` или `TEMP`).

46.6.4.6. Обработчик фильтрации источника

Необязательный обработчик `filter_by_origin_cb` вызывается, чтобы отметить, интересуют ли модуль вывода изменения, воспроизводимые из указанного источника (*`origin_id`*).

```
typedef bool (*LogicalDecodeFilterByOriginCB) (struct LogicalDecodingContext *ctx,
                                               RepOriginId origin_id);
```

В параметре *`ctx`* передаётся та же информация, что и для других обработчиков. Чтобы отметить, что изменения, поступающие из переданного узла, не представляют интереса, модуль должен вернуть true, вследствие чего эти изменения будут фильтроваться; в противном случае он должен вернуть false. Другие обработчики для фильтруемых транзакций и изменений вызываться не будут.

Это полезно при реализации каскадной или разнонаправленной репликации. Фильтрация по источнику в таких конфигурациях позволяет предотвратить передачу взад-вперёд одних и тех же изменений. Хотя информацию об источнике можно также извлечь из транзакций и изменений, фильтрация с помощью этого обработчика гораздо более эффективна.

46.6.4.7. Обработчик произвольных сообщений

Необязательный обработчик `message_cb` вызывается при получении сообщения логического декодирования.

```
typedef void (*LogicalDecodeMessageCB) (struct LogicalDecodingContext *ctx,
                                        ReorderBufferTXN *txn,
                                        XLogRecPtr message_lsn,
                                        bool transactional,
                                        const char *prefix,
                                        Size message_size,
                                        const char *message);
```

Параметр *`txn`* содержит метаинформацию о транзакции, включая время её фиксации и её XID. Заметьте, однако, что в нём может передаваться NULL, когда сообщение нетранзакционное и транзакции, в которой было выдано сообщение, ещё не назначен XID. В параметре *`lsn`* отмечается позиция сообщения в WAL. Параметр *`transactional`* показывает, было ли сообщение передано как транзакционное. В параметре *`prefix`* передаётся некоторый префикс (завершающийся нулём), по которому текущий модуль может выделять интересующие его сообщения. И наконец, параметр *`message`* содержит само сообщение размером *`message_size`* байт.

Необходимо дополнительно позаботиться о том, чтобы префикс, определяющий интересующие модуль вывода сообщения, был уникальным. Удачным выбором обычно будет имя расширения или самого модуля вывода.

46.6.5. Функции для формирования вывода

Чтобы действительно вывести данные, модули вывода могут записывать их в буфер `StringInfo` через `ctx->out`, внутри обработчиков `begin_cb`, `commit_cb` или `change_cb`. Прежде чем записывать данные в этот буфер, необходимо вызвать `OutputPluginPrepareWrite(ctx, last_write)`, а завершив запись в буфер, нужно вызвать `OutputPluginWrite(ctx, last_write)`, чтобы собственно произвести запись. Параметр *`last_write`* указывает, была ли эта определённая операция записи последней в данном обработчике.

Следующий пример показывает, как вывести данные для потребителя модуля вывода:

```
OutputPluginPrepareWrite(ctx, true);
appendStringInfo(ctx->out, "BEGIN %u", txn->xid);
OutputPluginWrite(ctx, true);
```

46.7. Запись вывода логического декодирования

Архитектура сервера позволяет добавлять другие методы вывода для логического декодирования. За подробностями обратитесь к коду `src/backend/replication/logical/logicalfuncs.c`. По сути, необходимо реализовать три функции: одну для чтения WAL, другую для подготовки к записи и третью для записи вывода (см. Подраздел 46.6.5).

46.8. Поддержка синхронной репликации для логического декодирования

46.8.1. Обзор

Логическое декодирование может использоваться для реализации синхронной репликации с тем же внешним интерфейсом, что и синхронная репликация поверх потоковой репликации. Для этого потоковая передача данных должна происходить через интерфейс потоковой репликации (см.

Раздел 46.3). Клиенты такой репликации должны посылать сообщения `Обновление состояния резервного сервера (F)` (см. Раздел 50.4), как и клиенты потоковой репликации.

> **Примечание**
>
> Синхронная реплика, получающая изменения через логическое декодирование, будет работать в рамках одной базы данных. Так как *synchronous_standby_names* в настоящее время, напротив, устанавливается на уровне сервера, это означает, что этот подход не будет работать корректно при использовании нескольких баз данных.

46.8.2. Ограничения

В схеме с логической репликацией возможна взаимоблокировка, если транзакция в исключительном режиме заблокирует таблицы каталога (в том числе пользовательские). Пользовательские таблицы каталога описаны в Подразделе 46.6.2. Это происходит, потому что в процессе логического декодирования транзакций при обращении к таблицам каталога они блокируются. Во избежание этого пользователи должны воздерживаться от исключительной блокировки таблиц каталога (в том числе пользовательских). Вызвать такую блокировку могут следующие обстоятельства:

- Установление явной блокировки (командой `LOCK`) для каталога `pg_class` в транзакции.
- Выполнение `CLUSTER` для каталога `pg_class` в транзакции.
- Выполнение `TRUNCATE` для таблицы каталога (в том числе пользовательской) в транзакции.

Заметьте, что эти команды, способные вызвать взаимоблокировку, могут применяться не только к явно указанным выше таблицам системного каталога, но также и к любым другим таблицам каталога (в том числе пользовательским).

Глава 47. Отслеживание прогресса репликации

Инфраструктура источников репликации введена для упрощения реализации решений логической репликации на основе логического декодирования. Она помогает решить две распространённых проблемы:

- Как надёжно отслеживать прогресс репликации
- Как менять поведение репликации в зависимости от источника строки; например, для предотвращения циклов при двунаправленной репликации

Источники репликации имеют только два свойства: имя и OID. Имя, по которому к источнику следует обращаться из разных систем, задаётся значением типа `text` в произвольной форме. Его следует выбирать так, чтобы конфликты между источниками репликации, созданными различными средствами репликации были маловероятны; например, добавлять в начало обозначение средства репликации. OID используется только для того, чтобы не приходилось хранить длинное имя там, где требуется минимизировать объём. Он не может разделяться между разными системами.

Источник репликации можно создать функцией `pg_replication_origin_create()`; удалить функцией `pg_replication_origin_drop()`; и увидеть в системном каталоге `pg_replication_origin`.

Одной из нетривиальных задач при организации репликации является надёжное отслеживание прогресса воспроизведения. Например, когда применяющий изменения процесс (или весь кластер) умирает, нужно иметь возможность понять, какие данные были переданы успешно. Наивные решения этой проблемы, такие как изменение строки в некоторой таблице для каждой воспроизведённой транзакции, чреваты дополнительной нагрузкой во время выполнения и замусориванием базы данных.

С использованием инфраструктуры источников репликации сеанс может быть помечен как воспроизводящий изменения с удалённого узла (с помощью функции `pg_replication_origin_session_setup()`). В дополнение к этому для каждой транзакции из источника можно задать LSN и время фиксации, вызвав `pg_replication_origin_xact_setup()`. Если сделать всё это, прогресс репликации можно будет отслеживать надёжным образом. Прогресс воспроизведения для всех источников репликации можно увидеть в представлении `pg_replication_origin_status`. Прогресс отдельного источника, например, при возобновлении репликации, можно получить для любого источника, воспользовавшись функцией `pg_replication_origin_progress()`, или для источника, настроенного в текущем сеансе, с помощью `pg_replication_origin_session_progress()`.

В топологиях репликации более сложных, чем простая репликация с одной системы в другую, возможна ещё одна проблема — повторная репликация уже воспроизведённых строк, что может приводить к зацикливанию и снижению эффективности. В качестве механизма выявления и предотвращения повторной репликации так же могут оказаться полезны источники репликации. Если воспользоваться функциями, упомянутыми в предыдущем абзаце, во все поступающие в сеансе транзакции и изменения, передаваемые обработчикам модулей вывода (см. Раздел 46.6), добавляется пометка источника репликации для текущего сеанса. Это позволяет обрабатывать их в модуле вывода по-разному, например, игнорировать все строки, кроме имеющих локальное происхождение. Кроме того, обработчик вызова `filter_by_origin_cb` позволяет отфильтровать поток изменений логического декодирования в зависимости от источника. Фильтрация через этот обработчик не так гибка, как проверка записей внутри модуля вывода, но зато гораздо эффективнее.

Часть VI. Справочное руководство

Статьи этого справочного руководства составлены так, чтобы дать в разумном объёме авторитетную, полную и формальную сводку по соответствующим темам. Дополнительные сведения об использовании Postgres Pro в повествовательной, ознакомительной или показательной форме можно найти в других частях этой книги. Ссылки на них можно найти на страницах этого руководства.

Все эти справочные статьи также публикуются в виде традиционных страниц «man».

Команды SQL

Эта часть документации содержит справочную информацию по командам SQL, поддерживаемым Postgres Pro. Под «SQL» здесь понимается язык вообще; сведения о соответствии стандартам и совместимости всех команд приведены на соответствующих страниц справочника.

ABORT

ABORT — прервать текущую транзакцию

Синтаксис

```
ABORT [ WORK | TRANSACTION ]
```

Описание

`ABORT` откатывает текущую транзакцию и приводит к отмене всех изменений, внесённых транзакцией. Эта команда ведёт себя так же, как и стандартная SQL-команда ROLLBACK, и существует только по историческим причинам.

Параметры

`WORK`
`TRANSACTION`

Необязательные ключевые слова, не оказывают никакого влияния.

Замечания

Чтобы завершить и зафиксировать транзакцию, используйте COMMIT.

При выполнении команды `ABORT` вне блока транзакции выдаётся предупреждение и больше ничего не происходит.

Примеры

Чтобы прервать все операции:

```
ABORT;
```

Совместимость

Эта команда является расширением Postgres Pro и существует по историческим причинам. Ей равнозначна стандартная SQL-команда `ROLLBACK`.

См. также

BEGIN, COMMIT, ROLLBACK

ALTER AGGREGATE

ALTER AGGREGATE — изменить определение агрегатной функции

Синтаксис

```
ALTER AGGREGATE имя ( сигнатура_агр_функции ) RENAME TO новое_имя
ALTER AGGREGATE имя ( сигнатура_агр_функции )
                OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }
ALTER AGGREGATE имя ( сигнатура_агр_функции ) SET SCHEMA новая_схема

Здесь сигнатура_агр_функции:

* |
[ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] |
[ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] ] ORDER BY
 [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ]
```

Описание

`ALTER AGGREGATE` изменяет определение агрегатной функции.

Выполнить `ALTER AGGREGATE` может только владелец соответствующей агрегатной функции. Чтобы сменить схему агрегатной функции, необходимо также иметь право `CREATE` в новой схеме. Чтобы сменить владельца, требуется также быть непосредственным или опосредованным членом новой роли, а эта роль должна иметь право `CREATE` в схеме агрегатной функции. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать агрегатную функцию. Однако суперпользователь может сменить владельца агрегатной функции в любом случае.)

Параметры

имя

Имя существующей агрегатной функции (возможно, дополненное схемой).

режим_аргумента

Режим аргумента: `IN` или `VARIADIC`. По умолчанию подразумевается `IN`.

имя_аргумента

Имя аргумента. Заметьте, что на самом деле `ALTER AGGREGATE` не обращает внимание на имена аргументов, так как для однозначной идентификации агрегатной функции достаточно только типов аргументов.

тип_аргумента

Тип входных данных, с которыми работает агрегатная функция. Чтобы сослаться на агрегатную функцию без аргументов, укажите вместо списка аргументов `*`, а чтобы сослаться на сортирующую агрегатную функцию, добавьте `ORDER BY` между указаниями непосредственных и агрегируемых аргументов.

новое_имя

Новое имя агрегатной функции.

новый_владелец

Новый владелец агрегатной функции.

новая_схема

Новая схема агрегатной функции.

Замечания

Если вы хотите сослаться на сортирующую агрегатную функцию, рекомендуется добавить `ORDER BY` между непосредственными и агрегируемыми аргументами так же, как и в CREATE AGGREGATE. Однако команда сработает и без `ORDER BY`, если непосредственные и агрегирующие аргументы перечислены подряд в одном списке. В такой сокращённой форме, если и в списке непосредственных, и в списке агрегирующих аргументов содержится `VARIADIC "any"`, достаточно написать `VARIADIC "any"` только один раз.

Примеры

Переименование агрегатной функции `myavg` для типа `integer` в `my_average`:

```
ALTER AGGREGATE myavg(integer) RENAME TO my_average;
```

Смена владельца агрегатной функции `myavg` для типа `integer` на `joe`:

```
ALTER AGGREGATE myavg(integer) OWNER TO joe;
```

Перемещение сортирующей агрегатной функции `mypercentile` с непосредственным аргументом типа `float8` и агрегируемым аргументом типа `integer` в схему `myschema`:

```
ALTER AGGREGATE mypercentile(float8 ORDER BY integer) SET SCHEMA myschema;
```

Это тоже будет работать:

```
ALTER AGGREGATE mypercentile(float8, integer) SET SCHEMA myschema;
```

Совместимость

Оператор `ALTER AGGREGATE` отсутствует в стандарте SQL.

См. также

CREATE AGGREGATE, DROP AGGREGATE

ALTER COLLATION

ALTER COLLATION — изменить определение правила сортировки

Синтаксис

```
ALTER COLLATION имя REFRESH VERSION

ALTER COLLATION имя RENAME TO новое_имя
ALTER COLLATION имя OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }
ALTER COLLATION имя SET SCHEMA новая_схема
```

Описание

`ALTER COLLATION` изменяет определение правила сортировки.

Выполнить `ALTER COLLATION` может только владелец соответствующего правила сортировки. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме правила сортировки. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать правило сортировки. Однако суперпользователь может сменить владельца правила сортировки в любом случае.)

Параметры

имя

Имя существующего правила сортировки (возможно, дополненное схемой).

новое_имя

Новое имя правила сортировки.

новый_владелец

Новый владелец правила сортировки.

новая_схема

Новая схема правила сортировки.

`REFRESH VERSION`

Обновить версию правила сортировки. Подробнее см. Раздел «Замечания».

Замечания

Когда применяются правила сортировки, предоставляемые библиотекой ICU, внутренняя версия сортировщика ICU записывается в системный каталог при создании объекта для данного правила. Когда такое правило используется, текущая версия сверяется с записанной и в случае несовпадения выдаётся предупреждение, например такое:

```
WARNING:  collation "xx-x-icu" has version mismatch
DETAIL:  The collation in the database was created using version 1.2.3.4, but the
 operating system provides version 2.3.4.5.
HINT:  Rebuild all objects affected by this collation and run ALTER COLLATION
 pg_catalog."xx-x-icu" REFRESH VERSION, or build Postgres Pro with the right library
 version.
```

Изменения в определениях правил сортировки могут приводить к разрушению индексов и другим проблемам, так как СУБД рассчитывает на то, что хранимые объекты отсортированы в

определённом порядке. Вообще этого следует избегать, но это может иметь место в совершенно легальных обстоятельствах, например при обновлении с помощью `pg_upgrade` исполняемых файлов сервера, скомпонованных с более новой версией ICU. Когда возникает такая ситуация, все объекты, зависящие от данного правила сортировки, должны быть перестроены, например, командой `REINDEX`. После этой операции можно обновить версию правила сортировки, выполнив команду `ALTER COLLATION ... REFRESH VERSION`. При этом системный каталог будет обновлён, в него будет записана текущая версия сортировщика, и предупреждение уйдёт. Заметьте, что эта команда собственно не проверяет, были ли все зависимые объекты перестроены корректно.

Следующий запрос позволяет выбрать все правила сортировки в текущей базе данных, которые требуют обновления, и зависящие от них объекты:

```
SELECT pg_describe_object(refclassid, refobjid, refobjsubid) AS "Collation",
       pg_describe_object(classid, objid, objsubid) AS "Object"
  FROM pg_depend d JOIN pg_collation c
       ON refclassid = 'pg_collation'::regclass AND refobjid = c.oid
  WHERE c.collversion <> pg_collation_actual_version(c.oid)
  ORDER BY 1, 2;
```

Примеры

Изменение названия правила сортировки с `ru_RU` на `russian`:

```
ALTER COLLATION "ru_RU" RENAME TO russian;
```

Изменение владельца правила сортировки `en_US` на `joe`:

```
ALTER COLLATION "en_US" OWNER TO joe;
```

Совместимость

Оператор `ALTER COLLATION` отсутствует в стандарте SQL.

См. также

CREATE COLLATION, DROP COLLATION

ALTER CONVERSION

ALTER CONVERSION — изменить определение перекодировки

Синтаксис

```
ALTER CONVERSION имя RENAME TO новое_имя
ALTER CONVERSION имя OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }
ALTER CONVERSION имя SET SCHEMA новая_схема
```

Описание

`ALTER CONVERSION` изменяет определение перекодировки.

Выполнить `ALTER CONVERSION` может только владелец соответствующей перекодировки. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме перекодировки. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать перекодировку. Однако суперпользователь может сменить владельца перекодировки в любом случае.)

Параметры

имя

Имя существующей перекодировки (возможно, дополненное схемой).

новое_имя

Новое имя перекодировки.

новый_владелец

Новый владелец перекодировки.

новая_схема

Новая схема перекодировки.

Примеры

Переименование перекодировки `iso_8859_1_to_utf8` в `latin1_to_unicode`:

```
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO latin1_to_unicode;
```

Смена владельца перекодировки `iso_8859_1_to_utf8` на `joe`:

```
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
```

Совместимость

Оператор `ALTER CONVERSION` отсутствует в стандарте SQL.

См. также

CREATE CONVERSION, DROP CONVERSION

ALTER DATABASE

ALTER DATABASE — изменить атрибуты базы данных

Синтаксис

```
ALTER DATABASE имя [ [ WITH ] параметр [ ... ] ]

Здесь параметр:

    ALLOW_CONNECTIONS разр_подключения
    CONNECTION LIMIT предел_подключений
    IS_TEMPLATE это_шаблон

ALTER DATABASE имя RENAME TO новое_имя

ALTER DATABASE имя OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }

ALTER DATABASE имя SET TABLESPACE новое_табл_пространство

ALTER DATABASE имя SET параметр_конфигурации { TO | = } { значение | DEFAULT }
ALTER DATABASE имя SET параметр_конфигурации FROM CURRENT
ALTER DATABASE имя RESET параметр_конфигурации
ALTER DATABASE имя RESET ALL
```

Описание

`ALTER DATABASE` изменяет атрибуты базы данных.

Первая форма оператора меняет параметры на уровне базы данных. (Подробнее описано ниже.) Изменять эти параметры может только владелец базы данных или суперпользователь.

Вторая форма меняет имя базы данных. Переименовать базу данных может только владелец БД или суперпользователь (не суперпользователю требуется также право `CREATEDB`). Переименовать текущую базу данных нельзя. (Если вам нужно сделать это, сначала подключитесь к другой базе.)

Третья форма меняет владельца базы данных. Чтобы сменить владельца базы, необходимо быть её владельцем и также непосредственным или опосредованным членом новой роли-владельца, и кроме того, иметь право `CREATEDB`. (Заметьте, что суперпользователи наделяются всеми этими правами автоматически.)

Четвёртая форма меняет табличное пространство по умолчанию для базы данных. Произвести это изменение может только её владелец или суперпользователь; кроме того, необходимо иметь право для создания нового табличного пространства. Эта команда физически переносит все таблицы или индексы из прежнего основного табличного пространства БД в новое. Новое табличное пространство должно быть пустым для этой базы данных и к ней никто не должен быть подключён. Таблицы и индексы, находящиеся не в основном табличном пространстве, при этом не затрагиваются.

Остальные формы меняют значение по умолчанию конфигурационных переменных времени выполнения для базы данных Postgres Pro. Когда устанавливается следующий сеанс работы с указанной базой данных, заданное этой командой значение становится значением по умолчанию. Значения переменных, заданные для базы данных, переопределяют значения, определённые в `postgresql.conf` или полученные через командную строку `postgres`. Менять сеансовые значения переменных для базы данных может только её владелец или суперпользователь. Некоторые параметры изменить таким образом нельзя, а некоторые может изменить только суперпользователь.

Параметры

имя

Имя базы данных, атрибуты которой изменяются.

разр_подключения

Если false, никто не сможет подключаться к этой базе данных.

предел_подключений

Число разрешённых одновременно подключений к этой базе данных (-1 снимает ограничение).

это_шаблон

Если true, базу данных сможет клонировать любой пользователь с правами `CREATEDB`; в противном случае клонировать эту базу смогут только суперпользователи и её владелец.

новое_имя

Новое имя базы данных.

новый_владелец

Новый владелец базы данных.

новое_табл_пространство

Новое основное табличное пространство базы данных.

Эту форму команды нельзя выполнять внутри блока транзакции.

параметр_конфигурации
значение

Устанавливает сеансовое значение по умолчанию для указанного параметра конфигурации. Если указывается *значение* `DEFAULT` или равнозначный вариант, `RESET`, определение параметра на уровне базы данных удаляется, так что в новых сеансах будет действовать значение по умолчанию, определённое на уровне системы. Для очистки всех значений параметров на уровне базы данных выполните `RESET ALL`. `SET FROM CURRENT` устанавливает значение параметра на уровне базы данных из текущего значения в активном сеансе.

За подробными сведениями об именах и значениях параметров обратитесь к SET и Главе 18.

Замечания

Также возможно связать параметры сеанса не с базой данных, а с определённой ролью; см. ALTER ROLE. В случае конфликта параметры на уровне роли переопределяют параметры на уровне базы данных.

Примеры

Отключение сканирования индекса по умолчанию в базе данных `test`:

```
ALTER DATABASE test SET enable_indexscan TO off;
```

Совместимость

Оператор `ALTER DATABASE` является расширением Postgres Pro.

См. также

CREATE DATABASE, DROP DATABASE, SET, CREATE TABLESPACE

ALTER DEFAULT PRIVILEGES

ALTER DEFAULT PRIVILEGES — определить права доступа по умолчанию

Синтаксис

```
ALTER DEFAULT PRIVILEGES
    [ FOR { ROLE | USER } целевая_роль [, ...] ]
    [ IN SCHEMA имя_схемы [, ...] ]
    предложение_GRANT_или_REVOKE

Где предложение_GRANT_или_REVOKE может быть следующим:

GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    TO { [ GROUP ] имя_роли | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    TO { [ GROUP ] имя_роли | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTIONS
    TO { [ GROUP ] имя_роли | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    TO { [ GROUP ] имя_роли | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | CREATE | ALL [ PRIVILEGES ] }
    ON SCHEMAS
    TO { [ GROUP ] имя_роли | PUBLIC } [, ...] [ WITH GRANT OPTION ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTIONS
    FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
```

```
    ON TYPES
    FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | CREATE | ALL [ PRIVILEGES ] }
    ON SCHEMAS
    FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

Описание

`ALTER DEFAULT PRIVILEGES` позволяет задавать права, применяемые к объектам, которые будут создаваться в будущем. (Эта команда не затрагивает права, назначенные уже существующим объектам.) В настоящее время можно задавать права только для схем, таблиц (включая представления и сторонние таблицы), последовательностей, функций и типов (включая домены).

Вы можете изменить права по умолчанию только для объектов, которые будут созданы вами или ролями, членами которых вы являетесь. Права можно задать глобально (т. е. для всех объектов, создаваемых в текущей базе данных) или для определённых схем.

Как объясняется в GRANT, права по умолчанию для объектов любого типа обычно дают все назначаемые разрешения владельцу объекта, а также могут давать некоторые разрешения роли `PUBLIC`. Однако это поведение можно поменять, изменив права по умолчанию командой `ALTER DEFAULT PRIVILEGES`.

Заданные на уровне схемы права по умолчанию добавляются к тем, что определены глобально для конкретного типа объекта. Это означает, что вы не можете отозвать права уровня схемы, если они назначены глобально (либо по умолчанию, либо предыдущей командой `ALTER DEFAULT PRIVILEGES` без указания схемы). Команда `REVOKE` для схемы может быть полезна только для отмены действия предыдущей команды `GRANT` для этой же схемы.

Параметры

целевая_роль

Имя существующей роли, членом которой является текущая. Если `FOR ROLE` опущено, подразумевается текущая роль.

имя_схемы

Имя существующей схемы. Если указано, права по умолчанию меняются для объектов, которые будут созданы в этой схеме. Если `IN SCHEMA` опущено, меняются глобальные права по умолчанию. Указание `IN SCHEMA` не допускается при установлении прав для схем, так как схемы не могут быть вложенными.

имя_роли

Имя существующей роли, для которой даются или отзываются права. Этот и все другие параметры в *предложении_grant_или_revoke* действуют как описано в GRANT или REVOKE, за исключением того, что они распространяются не на один конкретный объект, а на целый класс объектов.

Замечания

Чтобы узнать текущие назначенные права по умолчанию, воспользуйтесь командой `\ddp` в psql. Интерпретация значений прав приведена в описании команды `\dp` в GRANT.

Если вы желаете удалить роль, права по умолчанию для которой были изменены, необходимо явно отменить изменения прав по умолчанию или воспользоваться командой `DROP OWNED BY` для избавления от назначенных для этой роли прав по умолчанию.

Примеры

Наделение всех правом SELECT для всех таблиц (и представлений), которые будут созданы в дальнейшем в схеме `myschema`, и наделение роли `webuser` правом INSERT для этих же таблиц:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT ON TABLES TO PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT INSERT ON TABLES TO webuser;
```

Отмена предыдущих изменений с тем, чтобы для таблиц, создаваемых в будущем, были определены только обычные права, без дополнительных:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE SELECT ON TABLES FROM PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE INSERT ON TABLES FROM webuser;
```

Лишение роли public права на выполнение (EXECUTE), которое обычно даётся для функций (для всех функций, которые будут созданы ролью `admin`):

```
ALTER DEFAULT PRIVILEGES FOR ROLE admin REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

Однако заметьте, что этого же эффекта *нельзя* добиться с помощью команды, ограниченной одной схемой. Эта команда будет действовать, только если ей предшествовала соответствующая команда `GRANT`:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

Это объясняется тем, что на уровне схемы можно только назначить права по умолчанию, которые добавятся к назначенным глобально, но нельзя отозвать последние.

Совместимость

Оператор `ALTER DEFAULT PRIVILEGES` отсутствует в стандарте SQL.

См. также

GRANT, REVOKE

ALTER DOMAIN

ALTER DOMAIN — изменить определение домена

Синтаксис

```
ALTER DOMAIN имя
    { SET DEFAULT выражение | DROP DEFAULT }
ALTER DOMAIN имя
    { SET | DROP } NOT NULL
ALTER DOMAIN имя
    ADD ограничение_домена [ NOT VALID ]
ALTER DOMAIN имя
    DROP CONSTRAINT [ IF EXISTS ] имя_ограничения [ RESTRICT | CASCADE ]
ALTER DOMAIN имя
     RENAME CONSTRAINT имя_ограничения TO имя_нового_ограничения
ALTER DOMAIN имя
    VALIDATE CONSTRAINT имя_ограничения
ALTER DOMAIN имя
    OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }
ALTER DOMAIN имя
    RENAME TO новое_имя
ALTER DOMAIN имя
    SET SCHEMA новая_схема
```

Описание

`ALTER DOMAIN` изменяет определение существующего домена. Эта команда имеет несколько разновидностей:

`SET/DROP DEFAULT`

Эти формы задают/убирают значение по умолчанию для домена. Заметьте, что эти значения по умолчанию применяются только при последующих командах `INSERT`; они не меняются в строках с данным доменом, уже добавленных в таблицу.

`SET/DROP NOT NULL`

Эти формы определяют, будет ли домен принимать значения NULL или нет. `SET NOT NULL` можно выполнить, только если столбцы с этим доменом ещё не содержат значений NULL.

`ADD ограничение_домена [ NOT VALID ]`

Эта форма добавляет новое ограничение для домена с тем же синтаксисом, что описан в CREATE DOMAIN. Когда добавляется новое ограничение домена, все столбцы с этим доменом будут проверены на соответствие этому ограничению. Эти проверки можно подавить, добавив указание `NOT VALID`, а затем активировать позднее с помощью команды `ALTER DOMAIN ... VALIDATE CONSTRAINT`. Вновь вставленные или изменённые строки всегда проверяются по всем ограничениям, даже тем, что отмечены как `NOT VALID`. Указание `NOT VALID` допускается только для ограничений `CHECK`.

`DROP CONSTRAINT [ IF EXISTS ]`

Эта форма убирает ограничения домена. Если указано `IF EXISTS` и заданное ограничение не существует, это не считается ошибкой. В этом случае выдаётся только замечание.

`RENAME CONSTRAINT`

Эта форма меняет название ограничения домена.

VALIDATE CONSTRAINT

Эта форма включает проверку ограничения, ранее добавленного как NOT VALID, то есть проверяет все значения в столбцах с этим типом домена на соответствие этому ограничению.

OWNER

Эта форма меняет владельца домена на заданного пользователя.

RENAME

Эта форма меняет название домена.

SET SCHEMA

Эта форма меняет схему домена. Все ограничения, связанные с данным доменом, так же переносятся в новую схему.

Выполнить ALTER DOMAIN может только владелец соответствующего домена. Чтобы сменить схему домена, необходимо также иметь право CREATE в новой схеме. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право CREATE в схеме домена. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать домен. Однако суперпользователь может сменить владельца домена в любом случае.)

Параметры

имя

Имя существующего домена (возможно, дополненное схемой), подлежащего изменению.

ограничение_домена

Новое ограничение домена.

имя_ограничения

Имя существующего ограничения, подлежащего удалению или переименованию.

NOT VALID

Не проверять существующие сохранённые данные на соответствие ограничению.

CASCADE

Автоматически удалять объекты, зависящие от данного ограничения, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

RESTRICT

Отказать в удалении ограничения, если существуют зависящие от него объекты. Это поведение по умолчанию.

новое_имя

Новое имя домена.

имя_нового_ограничения

Новое имя ограничения.

новый_владелец

Имя пользователя, назначаемого новым владельцем домена.

новая_схема

Новая схема домена.

Замечания

Хотя команда `ALTER DOMAIN ADD CONSTRAINT` пытается проверить, удовлетворяют ли уже существующие данные новому ограничению, эта проверка не очень надёжна, так как данная команда не «видит» строки таблицы, которые были только что добавлены или изменены, но ещё не зафиксированы. Если есть риск того, что параллельные операции могут вставить неподходящие данные, можно применить следующий подход: создать ограничение с указанием `NOT VALID`, зафиксировать эту команду, подождать окончания всех транзакций, начатых до фиксирования, а затем выполнить `ALTER DOMAIN VALIDATE CONSTRAINT` для проведения контроля данных. В этом случае проверка будет надёжной, так как после фиксирования ограничения оно будет гарантированно действовать на все новые значения типа домена, вносимые последующими транзакциями.

В настоящее время команды `ALTER DOMAIN ADD CONSTRAINT`, `ALTER DOMAIN VALIDATE CONSTRAINT` и `ALTER DOMAIN SET NOT NULL` выдают ошибку, если проверяемый указанный домен или производный от него используется в столбце составного типа в любой таблице БД. В дальнейшем они будут доработаны, с тем чтобы новое ограничение проверялось и при такой вложенности.

Примеры

Добавление ограничения `NOT NULL` к домену:

```
ALTER DOMAIN zipcode SET NOT NULL;
```

Удаление ограничения `NOT NULL` из домена:

```
ALTER DOMAIN zipcode DROP NOT NULL;
```

Добавление ограничения-проверки к домену:

```
ALTER DOMAIN zipcode ADD CONSTRAINT zipchk CHECK (char_length(VALUE) = 5);
```

Удаление ограничения-проверки из домена:

```
ALTER DOMAIN zipcode DROP CONSTRAINT zipchk;
```

Переименование ограничения-проверки в домене:

```
ALTER DOMAIN zipcode RENAME CONSTRAINT zipchk TO zip_check;
```

Перемещение домена в другую схему:

```
ALTER DOMAIN zipcode SET SCHEMA customers;
```

Совместимость

`ALTER DOMAIN` соответствует стандарту SQL, за исключением подвидов `OWNER`, `RENAME`, `SET SCHEMA` и `VALIDATE CONSTRAINT`, которые являются расширениями Postgres Pro. Предложение `NOT VALID` вариации `ADD CONSTRAINT` также является расширением Postgres Pro.

См. также

CREATE DOMAIN, DROP DOMAIN

ALTER EVENT TRIGGER

ALTER EVENT TRIGGER — изменить определение событийного триггера

Синтаксис

```
ALTER EVENT TRIGGER имя DISABLE
ALTER EVENT TRIGGER имя ENABLE [ REPLICA | ALWAYS ]
ALTER EVENT TRIGGER имя OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }
ALTER EVENT TRIGGER имя RENAME TO новое_имя
```

Описание

`ALTER EVENT TRIGGER` изменяет свойства существующего событийного триггера.

Для изменения событийного триггера нужно быть суперпользователем.

Параметры

имя

Имя существующего триггера, подлежащего изменению.

новый_владелец

Имя пользователя, назначаемого новым владельцем событийного триггера.

новое_имя

Новое имя событийного триггера.

`DISABLE/ENABLE [ REPLICA | ALWAYS ] TRIGGER`

Эти формы настраивают срабатывание событийных триггеров. Отключённый триггер сохраняется в системе, но не выполняется, когда происходит его событие срабатывания. См. также session_replication_role.

Совместимость

Оператор `ALTER EVENT TRIGGER` отсутствует в стандарте SQL.

См. также

CREATE EVENT TRIGGER, DROP EVENT TRIGGER

ALTER EXTENSION

ALTER EXTENSION — изменить определение расширения

Синтаксис

```
ALTER EXTENSION имя UPDATE [ TO новая_версия ]
ALTER EXTENSION имя SET SCHEMA новая_схема
ALTER EXTENSION имя ADD элемент_объект
ALTER EXTENSION имя DROP элемент_объект

Здесь элемент_объект:

  ACCESS METHOD имя_объекта |
  AGGREGATE имя_агрегатной_функции ( сигнатура_агр_функции ) |
  CAST (исходный_тип AS целевой_тип) |
  COLLATION имя_объекта |
  CONVERSION имя_объекта |
  DOMAIN имя_объекта |
  EVENT TRIGGER имя_объекта |
  FOREIGN DATA WRAPPER имя_объекта |
  FOREIGN TABLE имя_объекта |
  FUNCTION имя_функции [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ] |
  MATERIALIZED VIEW имя_объекта |
  OPERATOR имя_оператора (тип_слева, тип_справа) |
  OPERATOR CLASS имя_объекта USING индексный_метод |
  OPERATOR FAMILY имя_объекта USING индексный_метод |
  [ PROCEDURAL ] LANGUAGE имя_объекта |
  SCHEMA имя_объекта |
  SEQUENCE имя_объекта |
  SERVER имя_объекта |
  TABLE имя_объекта |
  TEXT SEARCH CONFIGURATION имя_объекта |
  TEXT SEARCH DICTIONARY имя_объекта |
  TEXT SEARCH PARSER имя_объекта |
  TEXT SEARCH TEMPLATE имя_объекта |
  TRANSFORM FOR имя_типа LANGUAGE имя_языка |
  TYPE имя_объекта |
  VIEW имя_объекта

и сигнатура_агр_функции:

* |
[ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] |
[ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] ] ORDER BY
 [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ]
```

Описание

`ALTER EXTENSION` изменяет определение установленного расширения. Эта команда имеет несколько подвидов:

`UPDATE`

Эта форма обновляет версию расширения. Расширение должно предоставлять подходящий скрипт обновления (или набор скриптов), который может сменить текущую установленную версию на требуемую.

SET SCHEMA

Эта форма переносит объекты расширения в другую схему. Чтобы эта команда выполнилась успешно, расширение должно быть *перемещаемым*.

ADD *элемент_объект*

Эта форма добавляет существующий объект в расширение. В основном это применяется в скриптах обновления расширений. Добавленный объект затем будет считаться частью расширения, и удалить его можно будет, только удалив расширение.

DROP *элемент_объект*

Эта форма удаляет из расширения включённый в него объект. В основном это применяется в скриптах обновления расширений. Сам объект при этом не уничтожается, а только отделяется от расширения.

Подробнее эти операции описаны в Разделе 35.15.

Чтобы выполнить команду ALTER EXTENSION, необходимо быть владельцем данного расширения. Для форм ADD/DROP требуется также быть владельцем добавляемого/удаляемого объекта.

Параметры

имя

Имя установленного расширения.

новая_версия

Запрашиваемая новая версия расширения. Её можно записать в виде идентификатора или строкового значения. Если она не указана, ALTER EXTENSION UPDATE пытается выполнить обновление до версии, указанной в качестве версии по умолчанию в управляющем файле расширения.

новая_схема

Новая схема расширения.

имя_объекта
имя_агрегатной_функции
имя_функции
имя_оператора

Имя объекта, добавляемого или удаляемого из расширения. Имена таблиц, агрегатных функций, доменов, сторонних таблиц, функций, операторов, классов операторов, семейств операторов, последовательностей, объектов текстового поиска, типов и представлений можно дополнить именем схемы.

исходный_тип

Имя исходного типа данных для приведения.

целевой_тип

Имя целевого типа данных для приведения.

режим_аргумента

Режим аргумента обычной или агрегатной функции: IN, OUT, INOUT или VARIADIC. По умолчанию подразумевается IN. Заметьте, что ALTER EXTENSION не учитывает аргументы OUT, так как для идентификации функции нужны только типы входных аргументов. Поэтому достаточно перечислить только аргументы IN, INOUT и VARIADIC.

имя_аргумента

Имя аргумента обычной или агрегатной функции. Заметьте, что на самом деле `ALTER EXTENSION` не обращает внимание на имена аргументов, так как для однозначной идентификации функции достаточно только типов аргументов.

тип_аргумента

Тип данных аргумента обычной или агрегатной функции.

тип_слева
тип_справа

Тип данных аргументов оператора (возможно, дополненный именем схемы). В случае отсутствия аргумента префиксного или постфиксного оператора укажите вместо типа `NONE`.

`PROCEDURAL`

Это слово не несёт смысловой нагрузки.

имя_типа

Имя типа данных, для которого предназначена трансформация.

имя_языка

Имя языка, для которого предназначена трансформация.

Примеры

Обновление расширения `hstore` до версии 2.0:

```
ALTER EXTENSION hstore UPDATE TO '2.0';
```

Смена схемы расширения `hstore` на `utils`:

```
ALTER EXTENSION hstore SET SCHEMA utils;
```

Добавление существующей функции в расширение `hstore`:

```
ALTER EXTENSION hstore ADD FUNCTION populate_record(anyelement, hstore);
```

Совместимость

Оператор `ALTER EXTENSION` является расширением Postgres Pro.

См. также

CREATE EXTENSION, DROP EXTENSION

ALTER FOREIGN DATA WRAPPER

ALTER FOREIGN DATA WRAPPER — изменить определение обёртки сторонних данных

Синтаксис

```
ALTER FOREIGN DATA WRAPPER имя
    [ HANDLER функция_обработчик | NO HANDLER ]
    [ VALIDATOR функция_проверки | NO VALIDATOR ]
    [ OPTIONS ( [ ADD | SET | DROP ] параметр ['значение'] [, ... ]) ]
ALTER FOREIGN DATA WRAPPER имя OWNER TO { новый_владелец | CURRENT_USER |
 SESSION_USER }
ALTER FOREIGN DATA WRAPPER имя RENAME TO новое_имя
```

Описание

`ALTER FOREIGN DATA WRAPPER` изменяет определение обёртки сторонних данных. Первая форма команды меняет вспомогательные функции или общие параметры обёртки (требуется минимум одно предложение), а вторая — владельца обёртки.

Настраивать обёртки сторонних данных могут только суперпользователи и только суперпользователи могут быть их владельцами.

Параметры

имя

Имя существующей обёртки сторонних данных.

`HANDLER` *функция_обработчик*

Задаёт новое имя функции-обработчика для обёртки сторонних данных.

`NO HANDLER`

Эти ключевые слова указывают, что обёртка сторонних данных теперь не имеет функции-обработчика.

Заметьте, что обращаться к сторонним таблицам, если их обёртка сторонних данных не имеет обработчика, нельзя.

`VALIDATOR` *функция_проверки*

Задаёт новое имя функции проверки для обёртки сторонних данных.

Заметьте, что возможна ситуация, что предыдущие параметры обёртки данных, зависимых серверов, сопоставлений пользователей или сторонних таблиц окажутся неприемлемыми для новой функции проверки. Postgres Pro не проверяет их, поэтому пользователь сам должен убедиться в правильности этих параметров, прежде чем использовать изменённую обёртку данных. Однако параметры, изменяемые в данной команде `ALTER FOREIGN DATA WRAPPER`, будут проверены новой функцией проверки.

`NO VALIDATOR`

Эти ключевые слова указывают, что обёртка сторонних данных теперь не имеет функции проверки.

`OPTIONS ( [ ADD | SET | DROP ] параметр ['значение'] [, ... ] )`

Эта форма настраивает параметры обёртки сторонних данных. `ADD`, `SET` и `DROP` определяют, какое действие будет выполнено (добавление, установка и удаление, соответственно). Если

действие не задано явно, подразумевается `ADD`. Имена параметров должны быть уникальными, они вместе со значениями проверяются функцией проверки, если она установлена.

`новый_владелец`

Имя пользователя, назначаемого новым владельцем обёртки сторонних данных.

`новое_имя`

Новое имя обёртки сторонних данных.

Примеры

Изменение параметров обёртки сторонних данных `dbi`: добавление параметра `foo`, удаление `bar`:

```
ALTER FOREIGN DATA WRAPPER dbi OPTIONS (ADD foo '1', DROP 'bar');
```

Установление для обёртки сторонних данных `dbi` новой функции проверки `bob.myvalidator`:

```
ALTER FOREIGN DATA WRAPPER dbi VALIDATOR bob.myvalidator;
```

Совместимость

`ALTER FOREIGN DATA WRAPPER` соответствует стандарту ISO/IEC 9075-9 (SQL/MED), за исключением предложений `HANDLER`, `VALIDATOR`, `OWNER TO` и `RENAME`, являющихся расширениями.

См. также

CREATE FOREIGN DATA WRAPPER, DROP FOREIGN DATA WRAPPER

ALTER FOREIGN TABLE

ALTER FOREIGN TABLE — изменить определение сторонней таблицы

Синтаксис

```
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] имя [ * ]
    действие [, ... ]
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] имя [ * ]
    RENAME [ COLUMN ] имя_столбца TO новое_имя_столбца
ALTER FOREIGN TABLE [ IF EXISTS ] имя
    RENAME TO новое_имя
ALTER FOREIGN TABLE [ IF EXISTS ] имя
    SET SCHEMA новая_схема

Где действие может быть следующим:

    ADD [ COLUMN ] имя_столбца тип_данных [ COLLATE правило_сортировки ]
 [ ограничение_столбца [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] имя_столбца [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] имя_столбца [ SET DATA ] TYPE тип_данных
 [ COLLATE правило_сортировки ]
    ALTER [ COLUMN ] имя_столбца SET DEFAULT выражение
    ALTER [ COLUMN ] имя_столбца DROP DEFAULT
    ALTER [ COLUMN ] имя_столбца { SET | DROP } NOT NULL
    ALTER [ COLUMN ] имя_столбца SET STATISTICS integer
    ALTER [ COLUMN ] имя_столбца SET ( атрибут = значение [, ... ] )
    ALTER [ COLUMN ] имя_столбца RESET ( атрибут [, ... ] )
    ALTER [ COLUMN ] имя_столбца SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
    ALTER [ COLUMN ] имя_столбца OPTIONS ( [ ADD | SET | DROP ] параметр ['значение']
 [, ... ])
    ADD ограничение_таблицы [ NOT VALID ]
    VALIDATE CONSTRAINT имя_ограничения
    DROP CONSTRAINT [ IF EXISTS ]  имя_ограничения [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ имя_триггера | ALL | USER ]
    ENABLE TRIGGER [ имя_триггера | ALL | USER ]
    ENABLE REPLICA TRIGGER имя_триггера
    ENABLE ALWAYS TRIGGER имя_триггера
    SET WITH OIDS
    SET WITHOUT OIDS
    INHERIT таблица_родитель
    NO INHERIT таблица_родитель
    OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }
    OPTIONS ( [ ADD | SET | DROP ] параметр ['значение'] [, ... ])
```

Описание

`ALTER FOREIGN TABLE` меняет определение существующей сторонней таблицы. Эта команда имеет несколько разновидностей:

`ADD COLUMN`

Эта форма добавляет в стороннюю таблицу новый столбец, следуя тому же синтаксису, что и CREATE FOREIGN TABLE. В отличие от добавления столбца в обычную таблицу, при данной операции в базовом хранилище ничего не меняется; эта команда просто объявляет о доступности нового столбца через данную стороннюю таблицу.

`DROP COLUMN [ IF EXISTS ]`

Эта форма удаляет столбец из сторонней таблицы. Если что-либо зависит от этого столбца, например, представление, для успешного результата потребуется добавить `CASCADE`. Если указано `IF EXISTS` и этот столбец не существует, ошибка не происходит, вместо этого выдаётся замечание.

`SET DATA TYPE`

Эта форма меняет тип столбца сторонней таблицы. И это не влияет на нижележащее хранилище: данная операция просто меняет тип, который по мнению Postgres Pro будет иметь этот столбец.

`SET/DROP DEFAULT`

Эти формы задают или удаляют значение по умолчанию для столбцов. Значения по умолчанию применяются только при последующих командах `INSERT` или `UPDATE`; их изменения не отражаются в строках, уже существующих в таблице.

`SET/DROP NOT NULL`

Устанавливает, будет ли столбец принимать значения NULL или нет.

`SET STATISTICS`

Эта форма задаёт цель сбора статистики по столбцам для последующих операций ANALYZE. За подробностями обратитесь к описанию подобной формы ALTER TABLE.

`SET ( `*`атрибут = значение`*` [, ... ] )`
`RESET ( `*`атрибут`*` [, ... ] )`

Эта форма задаёт или сбрасывает значения атрибутов. За подробностями обратитесь к описанию подобной формы ALTER TABLE.

`SET STORAGE`

Эта форма задаёт режим хранения для столбца. За подробностями обратитесь к описанию подобной формы ALTER TABLE. Заметьте, что режим хранения не имеет значения, если обёртка сторонних данных для этой таблицы будет игнорировать его.

`ADD `*`ограничение_таблицы`*` [ NOT VALID ]`

Эта форма добавляет новое ограничение в стороннюю таблицу с применением того же синтаксиса, что и CREATE FOREIGN TABLE. В настоящее время поддерживаются только ограничения `CHECK`.

В отличие от ограничения, добавляемого для обычной таблицы, ограничение сторонней таблицы фактически никак не проверяется; эта команда сводится просто к заявлению о том, что все строки в сторонней таблице предположительно удовлетворяют новому условию. (Подробнее это рассматривается в описании CREATE FOREIGN TABLE.) Если ограничение помечено как `NOT VALID` (непроверенное), сервер не будет полагать, что оно выполняется; такая запись делается только на случай использования в будущем.

`VALIDATE CONSTRAINT`

Эта форма отмечает ограничение, которая ранее было помечено `NOT VALID`, как проверенное. Собственно для проверки этого ограничения ничего не делается, но последующие запросы будут полагать, что оно действует.

`DROP CONSTRAINT [ IF EXISTS ]`

Эта форма удаляет указанное ограничение сторонней таблицы. Если указано `IF EXISTS` и заданное ограничение не существует, это не считается ошибкой. В этом случае выдаётся только замечание.

`DISABLE/ENABLE [ REPLICA | ALWAYS ] TRIGGER`

Эти формы управляют триггерами, принадлежащими сторонней таблице. За подробностями обратитесь к описанию подобной формы ALTER TABLE.

`SET WITH OIDS`

Эта форма добавляет в таблицу системный столбец `oid` (см. Раздел 5.4). Если в таблице уже есть такой столбец, она не делает ничего. Обёртка сторонних данных должна поддерживать OID, иначе из этого столбца будут читаться просто нулевые значения.

Заметьте, что это не равнозначно команде `ADD COLUMN oid oid` (эта команда добавит не системный, а обычный столбец с подходящим именем `oid`).

`SET WITHOUT OIDS`

Эта форма удаляет из таблицы системный столбец `oid`. Это в точности равнозначно `DROP COLUMN oid RESTRICT`, за исключением того, что в случае отсутствия столбца `oid` ошибки не будет.

`INHERIT` *`таблица_родитель`*

Эта форма делает целевую стороннюю таблицу потомком указанной родительской таблицы. За подробностями обратитесь к описанию подобной формы ALTER TABLE.

`NO INHERIT` *`таблица_родитель`*

Эта форма удаляет целевую стороннюю таблицу из списка потомков указанной родительской таблицы.

`OWNER`

Эта форма меняет владельца сторонней таблицы на заданного пользователя.

`OPTIONS ( [ ADD | SET | DROP ]` *`параметр`* `['`*`значение`*`'] [, ... ] )`

Эта форма настраивает параметры сторонней таблицы или одного из её столбцов. `ADD`, `SET` и `DROP` определяют, какое действие будет выполнено (добавление, установка и удаление, соответственно). Если действие не задано явно, подразумевается `ADD`. Имена параметров не должны повторяться (хотя параметр таблицы и параметр столбца вполне могут иметь одно имя). Имена и значения параметров также проверяются библиотекой обёртки сторонних данных.

`RENAME`

Формы `RENAME` меняют имя сторонней таблицы или имя столбца в сторонней таблице.

`SET SCHEMA`

Эта форма переносит стороннюю таблицу в другую схему.

Все действия, кроме `RENAME` и `SET SCHEMA`, можно объединить в один список изменений и выполнить одновременно. Например, можно добавить несколько столбцов и/или изменить тип столбцов одной командой.

Если команда записана в виде `ALTER FOREIGN TABLE IF EXISTS ...` и сторонняя таблица не существует, это не считается ошибкой. В этом случае выдаётся только замечание.

Выполнить `ALTER FOREIGN TABLE` может только владелец соответствующей таблицы. Чтобы сменить схему сторонней таблицы, необходимо также иметь право `CREATE` в новой схеме. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме таблицы. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать таблицу. Однако суперпользователь может сменить владельца таблицы в любом случае.) Чтобы добавить столбец или изменить тип столбец, ещё требуется иметь право `USAGE` для его типа данных.

Параметры

имя

Имя (возможно, дополненное схемой) существующей сторонней таблицы, подлежащей изменению. Если перед именем таблицы указано ONLY, изменяется только заданная таблица. Без ONLY изменяется и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно также добавить необязательное указание *, чтобы явно обозначить, что изменению подлежат все дочерние таблицы.

имя_столбца

Имя нового или существующего столбца.

новое_имя_столбца

Новое имя существующего столбца.

новое_имя

Новое имя таблицы.

тип_данных

Тип данных нового столбца или новый тип данных существующего столбца.

ограничение_таблицы

Новое ограничение уровня таблицы для сторонней таблицы.

имя_ограничения

Имя существующего ограничения, подлежащего удалению.

CASCADE

Автоматически удалять объекты, зависящие от удаляемого столбца или ограничения (например, представления, содержащие этот столбец), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

RESTRICT

Отказать в удалении столбца или ограничения, если существуют зависящие от них объекты. Это поведение по умолчанию.

имя_триггера

Имя включаемого или отключаемого триггера.

ALL

Отключает или включает все триггеры, принадлежащие сторонней таблице. (Если какие-либо из триггеров являются внутрисистемными, для этого требуются права суперпользователя. Сама система не добавляет такие триггеры в сторонние таблицы, но дополнительный код может сделать это.)

USER

Отключает или включает все триггеры, принадлежащие сторонней таблице, кроме сгенерированных внутрисистемных.

таблица_родитель

Родительская таблица, с которой будет установлена или разорвана связь данной сторонней таблицы.

новый_владелец

Имя пользователя, назначаемого новым владельцем таблицы.

новая_схема

Имя схемы, в которую будет перемещена таблица.

Замечания

Ключевое слово `COLUMN` не несёт смысловой нагрузки и может быть опущено.

При добавлении или удалении столбцов (`ADD COLUMN`/`DROP COLUMN`), добавлении ограничений `NOT NULL` или `CHECK` или изменении типа данных (`SET DATA TYPE`) согласованность этих определений с внешним сервером не гарантируется. Ответственность за соответствие определений таблицы удалённой стороне лежит на пользователе.

За более полным описанием параметров обратитесь к CREATE FOREIGN TABLE.

Примеры

Установление ограничения NOT NULL для столбца:

```
ALTER FOREIGN TABLE distributors ALTER COLUMN street SET NOT NULL;
```

Изменение параметров сторонней таблицы:

```
ALTER FOREIGN TABLE myschema.distributors OPTIONS (ADD opt1 'value', SET opt2 'value2',
 DROP opt3 'value3');
```

Совместимость

Формы `ADD`, `DROP` и `SET DATA TYPE` соответствуют стандарту SQL. Другие формы являются собственными расширениями Postgres Pro. Кроме того, возможность указать в одной команде `ALTER FOREIGN TABLE` несколько операций так же является расширением.

`ALTER FOREIGN TABLE DROP COLUMN` позволяет удалить единственный столбец сторонней таблицы и оставить таблицу без столбцов. Это является расширением стандарта SQL, который не допускает существование сторонних таблиц с нулём столбцов.

См. также

CREATE FOREIGN TABLE, DROP FOREIGN TABLE

ALTER FUNCTION

ALTER FUNCTION — изменить определение функции

Синтаксис

```
ALTER FUNCTION имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    действие [ ... ] [ RESTRICT ]
ALTER FUNCTION имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    RENAME TO новое_имя
ALTER FUNCTION имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }
ALTER FUNCTION имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    SET SCHEMA новая_схема
ALTER FUNCTION имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    DEPENDS ON EXTENSION имя_расширения

Где действие может быть следующим:

    CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    IMMUTABLE | STABLE | VOLATILE
    [ NOT ] LEAKPROOF
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    PARALLEL { UNSAFE | RESTRICTED | SAFE }
    COST стоимость_выполнения
    ROWS строк_в_результате
    SET параметр_конфигурации { TO | = } { значение | DEFAULT }
    SET параметр_конфигурации FROM CURRENT
    RESET параметр_конфигурации
    RESET ALL
```

Описание

`ALTER FUNCTION` изменяет определение функции.

Выполнить `ALTER FUNCTION` может только владелец соответствующей функции. Чтобы сменить схему функции, необходимо также иметь право `CREATE` в новой схеме. Чтобы сменить владельца, требуется также быть непосредственным или опосредованным членом новой роли, а эта роль должна иметь право `CREATE` в схеме функции. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать функцию. Однако суперпользователь может сменить владельца функции в любом случае.)

Параметры

имя

Имя существующей функции (возможно, дополненное схемой). Если список аргументов не указан, это имя должно быть уникальным в схеме.

режим_аргумента

Режим аргумента: `IN`, `OUT`, `INOUT` или `VARIADIC`. По умолчанию подразумевается `IN`. Заметьте, что `ALTER FUNCTION` не учитывает аргументы `OUT`, так как для идентификации функции нужны

только типы входных аргументов. Поэтому достаточно перечислить только аргументы `IN`, `INOUT` и `VARIADIC`.

имя_аргумента

Имя аргумента. Заметьте, что на самом деле `ALTER FUNCTION` не обращает внимание на имена аргументов, так как для однозначной идентификации функции достаточно только типов аргументов.

тип_аргумента

Тип данных аргументов функции (возможно, дополненный именем схемы), если таковые имеются.

новое_имя

Новое имя функции.

новый_владелец

Новый владелец функции. Заметьте, что если функция помечена как `SECURITY DEFINER`, в дальнейшем она будет выполняться от имени нового владельца.

новая_схема

Новая схема функции.

имя_расширения

Имя расширения, от которого будет зависеть функция.

```
CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT
```

`CALLED ON NULL INPUT` меняет функцию так, чтобы она вызывалась, когда некоторые или все её аргументы равны NULL. `RETURNS NULL ON NULL INPUT` или `STRICT` меняет функцию так, чтобы она не вызывалась, когда некоторые или все её аргументы равны NULL, а вместо вызова автоматически выдавался результат NULL. За подробностями обратитесь к CREATE FUNCTION.

```
IMMUTABLE
STABLE
VOLATILE
```

Устанавливает заданный вариант изменчивости функции. Подробнее это описано в CREATE FUNCTION.

```
[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER
```

Устанавливает, является ли функция определяющей контекст безопасности. Ключевое слово `EXTERNAL` игнорируется для соответствия стандарту SQL. Подробнее это свойство описано в CREATE FUNCTION.

```
PARALLEL
```

Устанавливает, будет ли функция считаться безопасной для распараллеливания. Подробнее это описано в CREATE FUNCTION.

```
LEAKPROOF
```

Устанавливает, является ли функция герметичной. Подробнее это свойство описано в CREATE FUNCTION.

COST *стоимость_выполнения*

Изменяет ориентировочную стоимость выполнения функции. Подробнее это описывается в CREATE FUNCTION.

ROWS *строк_в_результате*

Изменяет ориентировочное число строк в результате функции, возвращающей множество. Подробнее это описывается в CREATE FUNCTION.

параметр_конфигурации
значение

Добавляет или изменяет установку параметра конфигурации, выполняемую при вызове функции. Если задано *значение* DEFAULT или, что равнозначно, выполняется действие RESET, локальное переопределение для функции удаляется и функция выполняется со значением, установленным в окружении. Для удаления всех установок параметров для данной функции укажите RESET ALL. SET FROM CURRENT устанавливает для последующих вызовов функции значение параметра, действующее в момент выполнения ALTER PROCEDURE.

За подробными сведениями об именах и значениях параметров обратитесь к SET и Главе 18.

RESTRICT

Игнорируется для соответствия стандарту SQL.

Примеры

Переименование функции sqrt для типа integer в square_root:

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

Смена владельца функции sqrt для типа integer на joe:

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

Смена схемы функции sqrt для типа integer на maths:

```
ALTER FUNCTION sqrt(integer) SET SCHEMA maths;
```

Обозначение функции sqrt для типа integer как зависимой от расширения mathlib:

```
ALTER FUNCTION sqrt(integer) DEPENDS ON EXTENSION mathlib;
```

Изменение пути поиска, который устанавливается автоматически для функции:

```
ALTER FUNCTION check_password(text) SET search_path = admin, pg_temp;
```

Отмена автоматического определения search_path для функции:

```
ALTER FUNCTION check_password(text) RESET search_path;
```

Теперь функция будет выполняться с тем путём, который задан в момент вызова.

Совместимость

Этот оператор частично совместим с оператором ALTER FUNCTION в стандарте SQL. Стандарт позволяет менять больше свойств функции, но не позволяет переименовывать функции, переключать контекст безопасности, связывать с функциями значения параметров конфигурации, а также менять владельца, схему и тип изменчивости функции. Также в стандарте слово RESTRICT считается обязательным, тогда как в Postgres Pro оно не требуется.

См. также

CREATE FUNCTION, DROP FUNCTION

ALTER GROUP

ALTER GROUP — изменить имя роли или членство

Синтаксис

```
ALTER GROUP указание_роли ADD USER имя_пользователя [, ... ]
ALTER GROUP указание_роли DROP USER имя_пользователя [, ... ]

Здесь указание_роли:

    имя_роли
  | CURRENT_USER
  | SESSION_USER

ALTER GROUP имя_группы RENAME TO новое_имя
```

Описание

`ALTER GROUP` изменяет атрибуты группы пользователей. Эта команда считается устаревшей, хотя и поддерживается для обратной совместимости, так как группы (и пользователи) были заменены более общей концепцией ролей.

Первые две формы добавляют пользователей в группу или удаляют их из группы. (В данном случае в качестве «пользователя» или «группы» может фигурировать любая роль.) По сути они равнозначны командам разрешающим/запрещающим членство в роли «группа»; поэтому вместо них рекомендуется использовать GRANT и REVOKE.

Третья форма меняет имя группы. Она в точности равнозначна команде ALTER ROLE, выполняющей переименование роли.

Параметры

имя_группы

Имя изменяемой группы (роли).

имя_пользователя

Пользователи (роли), добавляемые или исключаемые из группы. Эти пользователи должны уже существовать; `ALTER GROUP` не создаёт и не удаляет пользователей.

новое_имя

Новое имя группы.

Примеры

Добавление пользователей в группу:

```
ALTER GROUP staff ADD USER karl, john;
```

Удаление пользователей из группы:

```
ALTER GROUP workers DROP USER beth;
```

Совместимость

Оператор `ALTER GROUP` отсутствует в стандарте SQL.

См. также

GRANT, REVOKE, ALTER ROLE

ALTER INDEX

ALTER INDEX — изменить определение индекса

Синтаксис

```
ALTER INDEX [ IF EXISTS ] имя RENAME TO новое_имя
ALTER INDEX [ IF EXISTS ] имя SET TABLESPACE табл_пространство
ALTER INDEX имя DEPENDS ON EXTENSION имя_расширения
ALTER INDEX [ IF EXISTS ] имя SET ( параметр_хранения [= значение] [, ... ] )
ALTER INDEX [ IF EXISTS ] имя RESET ( параметр_хранения [, ... ] )
ALTER INDEX ALL IN TABLESPACE имя [ OWNED BY имя_роли [, ... ] ]
    SET TABLESPACE новое_табл_пространство [ NOWAIT ]
```

Описание

`ALTER INDEX` меняет определение существующего индекса. Эта команда имеет несколько разновидностей:

`RENAME`

Форма `RENAME` меняет имя индекса. На сохранённые данные это не влияет.

`SET TABLESPACE`

Эта форма меняет табличное пространство индекса на заданное и переносит в него файл(ы) данных, связанные с индексом. Для изменения табличного пространства индекса нужно быть владельцем индекса и иметь право `CREATE` в новом табличном пространстве. Форма `ALL IN TABLESPACE` позволяет перенести из заданного пространства все индексы в текущей базе данных, блокируя их для перемещения и затем перемещая каждый индекс. Эта форма также поддерживает указание `OWNED BY`, с которым будут перемещены только индексы, принадлежащие заданным ролям. Если указан параметр `NOWAIT`, команда завершится ошибкой, если не сможет немедленно получить все требуемые блокировки. Заметьте, что эта команда не переместит системные каталоги; вместо неё следует использовать `ALTER DATABASE` или явные вызовы `ALTER INDEX`. См. также CREATE TABLESPACE.

`DEPENDS ON EXTENSION`

Эта форма помечает индекс как зависимый от расширения, так что при удалении расширения будет автоматически удалён и индекс.

`SET ( параметр_хранения [= значение] [, ... ] )`

Эта форма настраивает один или несколько специфичных для индекса параметров хранения. Список доступных параметров приведён в CREATE INDEX. Заметьте, что эта команда не меняет содержимое индекса немедленно; для получения желаемого эффекта в зависимости от параметров может потребоваться перестроить индекс командой REINDEX.

`RESET ( параметр_хранения [, ... ] )`

Эта форма сбрасывает один или несколько специфичных для индекса параметров хранения к значениям по умолчанию. Как и с `SET`, для полного обновления индекса может потребоваться выполнить `REINDEX`.

Параметры

`IF EXISTS`

Не считать ошибкой, если индекс не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) существующего индекса, подлежащего изменению.

новое_имя

Новое имя индекса.

табл_пространство

Табличное пространство, в которое будет перемещён индекс.

имя_расширения

Имя расширения, от которого будет зависеть индекс.

параметр_хранения

Имя специфичного для индекса параметра хранения.

значение

Новое значение специфичного для индекса параметра хранения. Это может быть число или строка, в зависимости от параметра.

Замечания

Эти операции также возможно выполнить с помощью ALTER TABLE. На самом деле `ALTER INDEX` — это просто синоним нескольких форм `ALTER TABLE`, работающих с индексами.

Ранее существовала форма `ALTER INDEX OWNER`, но сейчас она игнорируется (с предупреждением). Владельцем индекса может быть только владелец соответствующей таблицы. При смене владельца таблицы владелец индекса меняется автоматически.

Какие-либо изменения индексов системного каталога не допускаются.

Примеры

Переименование существующего индекса:

```
ALTER INDEX distributors RENAME TO suppliers;
```

Перемещение индекса в другое табличное пространство:

```
ALTER INDEX distributors SET TABLESPACE fasttablespace;
```

Изменение фактора заполнения индекса (предполагается, что это поддерживает метод индекса):

```
ALTER INDEX distributors SET (fillfactor = 75);
REINDEX INDEX distributors;
```

Совместимость

`ALTER INDEX` является расширением Postgres Pro.

См. также

CREATE INDEX, REINDEX

ALTER LANGUAGE

ALTER LANGUAGE — изменить определение процедурного языка

Синтаксис

```
ALTER [ PROCEDURAL ] LANGUAGE имя RENAME TO новое_имя
ALTER [ PROCEDURAL ] LANGUAGE имя OWNER TO { новый_владелец | CURRENT_USER |
 SESSION_USER }
```

Описание

`ALTER LANGUAGE` изменяет определение процедурного языка. Единственное, что может это команда — переименовать язык или назначить нового владельца. Выполнить `ALTER LANGUAGE` может только суперпользователь или владелец языка.

Параметры

имя

Имя языка

новое_имя

Новое имя языка

новый_владелец

Новый владелец языка

Совместимость

Оператор `ALTER LANGUAGE` отсутствует в стандарте SQL.

См. также

CREATE LANGUAGE, DROP LANGUAGE

ALTER LARGE OBJECT

ALTER LARGE OBJECT — изменить определение большого объекта

Синтаксис

```
ALTER LARGE OBJECT oid_большого_объекта OWNER TO { новый_владелец | CURRENT_USER |
 SESSION_USER }
```

Описание

`ALTER LARGE OBJECT` изменяет определение большого объекта.

Выполнить `ALTER LARGE OBJECT` может только владелец большого объекта. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца. (Однако суперпользователь может изменять свойства больших объектов в любом случае.) В настоящее время единственное возможное изменение заключается в назначении нового владельца, так что всегда действуют оба ограничения.

Параметры

oid_большого_объекта

OID изменяемого большого объекта

новый_владелец

Новый владелец большого объекта

Совместимость

Оператор `ALTER LARGE OBJECT` отсутствует в стандарте SQL.

См. также

Глава 32

ALTER MATERIALIZED VIEW

ALTER MATERIALIZED VIEW — изменить определение материализованного представления

Синтаксис

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] имя
    действие [, ... ]
ALTER MATERIALIZED VIEW имя
    DEPENDS ON EXTENSION имя_расширения
ALTER MATERIALIZED VIEW [ IF EXISTS ] имя
    RENAME [ COLUMN ] имя_столбца TO новое_имя_столбца
ALTER MATERIALIZED VIEW [ IF EXISTS ] имя
    RENAME TO новое_имя
ALTER MATERIALIZED VIEW [ IF EXISTS ] имя
    SET SCHEMA новая_схема
ALTER MATERIALIZED VIEW ALL IN TABLESPACE имя [ OWNED BY имя_роли [, ... ] ]
    SET TABLESPACE новое_табл_пространство [ NOWAIT ]

Где действие может быть следующим:

    ALTER [ COLUMN ] имя_столбца SET STATISTICS integer
    ALTER [ COLUMN ] имя_столбца SET ( атрибут = значение [, ... ] )
    ALTER [ COLUMN ] имя_столбца RESET ( атрибут [, ... ] )
    ALTER [ COLUMN ] имя_столбца SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
    CLUSTER ON имя_индекса
    SET WITHOUT CLUSTER
    SET TABLESPACE новое_табл_пространство
    SET ( параметр_хранения [= значение] [, ... ] )
    RESET ( параметр_хранения [, ... ] )
    OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }
```

Описание

`ALTER MATERIALIZED VIEW` изменяет различные расширенные свойства существующего материализованного представления.

Выполнить `ALTER MATERIALIZED VIEW` может только владелец материализованного представления. Чтобы сменить схему материализованного представления, необходимо также иметь право `CREATE` в новой схеме. Чтобы сменить владельца, требуется также быть непосредственным или опосредованным членом новой роли, а эта роль должна иметь право `CREATE` в схеме материализованного представления. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать материализованное представление. Однако суперпользователь может сменить владельца материализованного представления в любом случае.)

Форма `DEPENDS ON EXTENSION` помечает материализованное представление как зависимое от расширения, так что матпредставление будет автоматически удаляться при удалении расширения.

Подвиды и действия оператора `ALTER MATERIALIZED VIEW` являются подмножеством тех, что относятся к команде `ALTER TABLE`, и имеют то же значение применительно к материализованным представлениям. За подробностями обратитесь к описанию ALTER TABLE.

Параметры

имя

Имя существующего материализованного представления (возможно, дополненное схемой).

имя_столбца

Имя нового или существующего столбца.

имя_расширения

Имя расширения, от которого будет зависеть материализованное представление.

новое_имя_столбца

Новое имя существующего столбца.

новый_владелец

Имя пользователя, назначаемого новым владельцем материализованного представления.

новое_имя

Новое имя материализованного представления.

новая_схема

Новая схема материализованного представления.

Примеры

Переименование материализованного представления `foo` в `bar`:

```
ALTER MATERIALIZED VIEW foo RENAME TO bar;
```

Совместимость

`ALTER MATERIALIZED VIEW` является расширением Postgres Pro.

См. также

CREATE MATERIALIZED VIEW, DROP MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

ALTER OPERATOR

ALTER OPERATOR — изменить определение оператора

Синтаксис

```
ALTER OPERATOR имя ( { тип_слева | NONE } , { тип_справа | NONE } )
    OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }

ALTER OPERATOR имя ( { тип_слева | NONE } , { тип_справа | NONE } )
    SET SCHEMA новая_схема

ALTER OPERATOR имя ( { тип_слева | NONE } , { тип_справа | NONE } )
    SET ( {  RESTRICT = { процедура_ограничения | NONE }
           | JOIN = { процедура_соединения | NONE }
         } [, ... ] )
```

Описание

`ALTER OPERATOR` изменяет определение оператора.

Выполнить `ALTER OPERATOR` может только владелец соответствующего оператора. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме оператора. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать оператор. Однако суперпользователь может сменить владельца оператора в любом случае.)

Параметры

имя

Имя существующего оператора (возможно, дополненное схемой).

тип_слева

Тип данных левого операнда оператора; если у оператора нет левого операнда, укажите `NONE`.

тип_справа

Тип данных правого операнда оператора; если у оператора нет правого операнда, укажите `NONE`.

новый_владелец

Новый владелец оператора.

новая_схема

Новая схема оператора.

процедура_ограничения

Функция оценки избирательности ограничения для данного оператора; значение NONE удаляет существующую функцию оценки.

процедура_соединения

Функция оценки избирательности соединения для этого оператора; значение NONE удаляет существующую функцию оценки.

Примеры

Смена владельца нестандартного оператора `a @@ b` для типа `text`:

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

Смена функций оценки избирательности ограничения и соединения для нестандартного оператора `a && b` для типа `int[]`:

```
ALTER OPERATOR && (_int4, _int4) SET (RESTRICT = _int_contsel, JOIN =
 _int_contjoinsel);
```

Совместимость

Команда `ALTER OPERATOR` отсутствует в стандарте SQL.

См. также

CREATE OPERATOR, DROP OPERATOR

ALTER OPERATOR CLASS

ALTER OPERATOR CLASS — изменить определение класса операторов

Синтаксис

```
ALTER OPERATOR CLASS имя USING индексный_метод
    RENAME TO новое_имя

ALTER OPERATOR CLASS имя USING индексный_метод
    OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }

ALTER OPERATOR CLASS имя USING индексный_метод
    SET SCHEMA новая_схема
```

Описание

`ALTER OPERATOR CLASS` изменяет определение класса операторов.

Выполнить `ALTER OPERATOR CLASS` может только владелец соответствующего класса операторов. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме класса операторов. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать класс операторов. Однако суперпользователь может сменить владельца классов операторов в любом случае.)

Параметры

имя

Имя существующего класса операторов (возможно, дополненное схемой).

индексный_метод

Имя индексного метода, для которого предназначен этот класс операторов.

новое_имя

Новое имя класса операторов.

новый_владелец

Новый владелец класса операторов.

новая_схема

Новая схема класса операторов.

Совместимость

Команда `ALTER OPERATOR CLASS` отсутствует в стандарте SQL.

См. также

CREATE OPERATOR CLASS, DROP OPERATOR CLASS, ALTER OPERATOR FAMILY

ALTER OPERATOR FAMILY

ALTER OPERATOR FAMILY — изменить определение семейства операторов

Синтаксис

```
ALTER OPERATOR FAMILY имя USING индексный_метод ADD
  {  OPERATOR номер_стратегии имя_оператора ( тип_операнда, тип_операнда )
             [ FOR SEARCH | FOR ORDER BY семейство_сортировки ]
   | FUNCTION номер_опорной_функции [ ( тип_операнда [ , тип_операнда ] ) ]
             имя_функции [ ( тип_аргумента [, ...] ) ]
  } [, ... ]

ALTER OPERATOR FAMILY имя USING индексный_метод DROP
  {  OPERATOR номер_стратегии ( тип_операнда [ , тип_операнда ] )
   | FUNCTION номер_опорной_функции ( тип_операнда [ , тип_операнда ] )
  } [, ... ]

ALTER OPERATOR FAMILY имя USING индексный_метод
    RENAME TO новое_имя

ALTER OPERATOR FAMILY имя USING индексный_метод
    OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }

ALTER OPERATOR FAMILY имя USING индексный_метод
    SET SCHEMA новая_схема
```

Описание

`ALTER OPERATOR FAMILY` меняет определение семейства операторов. Она позволяет добавлять в семейство операторы и опорные функции, удалять их из семейства или менять имя и владельца семейства операторов.

Когда операторы и опорные функции добавляются в семейство с помощью `ALTER OPERATOR FAMILY`, они не становятся частью какого-либо определённого класса операторов в семействе, а просто считаются «слабосвязанными» с семейством. Это показывает, что эти операторы и функции семантически совместимы с семейством, но не требуются для корректной работы какого-либо индекса. (Операторы и функции, которые действительно требуются для этого, должны быть включены не в семейство, а в класс операторов; см. CREATE OPERATOR CLASS.) Postgres Pro позволяет удалять слабосвязанные члены из семейства в любое время, но члены класса операторов не могут быть удалены, пока не будет удалён весь класс и все зависимые от него индексы. Обычно в классы операторов включаются операторы и функции, работающие с одним типом данным (так как они нужны для поддержки индексов данных такого типа), а функции и операторы, работающие с разными типами, становятся слабосвязанными членами семейства.

Выполнить `ALTER OPERATOR FAMILY` может только суперпользователь. (Это ограничение введено потому, что ошибочное определение семейства операторов может вызвать нарушения или даже сбой в работе сервера.)

`ALTER OPERATOR FAMILY` в настоящее время не проверяет, включает ли определение семейства операторов все операторы и функции, требуемые для индексного метода, и образуют ли они целостный набор. Ответственность за правильность определения семейства лежит на пользователе.

За дополнительными сведениями обратитесь к Разделу 35.14.

Параметры

имя

Имя существующего семейства операторов (возможно, дополненное схемой).

индексный_метод

Имя индексного метода, для которого предназначено это семейство операторов.

номер_стратегии

Номер стратегии индексного метода для оператора, связанного с данным семейством операторов.

имя_оператора

Имя (возможно, дополненное схемой) оператора, связанного с данным семейством операторов.

тип_операнда

В предложении `OPERATOR` указывается тип(ы) данных оператора или `NONE`, если это левый или правый унарный оператор. В отличие от похожего синтаксиса в `CREATE OPERATOR CLASS`, здесь типы операндов должны указываться всегда.

В предложении `ADD FUNCTION` это тип данных, который должна поддерживать эта функция, если он отличается от входного типа данных функции. Для функций сравнения B-деревьев и хеш-функций указывать *тип_операнда* необязательно, так как их входные типы данных всегда будут подходящими. Однако для функций поддержки сортировки B-деревьев и всех функций в классах операторов GiST, SP-GiST и GIN необходимо указать тип(ы) операндов, с которыми будут использоваться эти функции.

В предложении `DROP FUNCTION` тип операнда, который должна поддерживать эта функция.

семейство_сортировки

Имя (возможно, дополненное схемой) существующего семейства операторов `btree`, описывающего порядок сортировки, связанный с оператором сортировки.

Если не указано ни `FOR SEARCH` (для поиска), ни `FOR ORDER BY` (для сортировки), подразумевается `FOR SEARCH`.

номер_опорной_функции

Номер опорной процедуры индексного метода для функции, связанной с данным семейством операторов.

имя_функции

Имя (возможно, дополненное схемой) функции, которая является опорной процедурой индексного метода для данного семейства операторов. Если список аргументов отсутствует, имя функции должно быть уникальным в её схеме.

тип_аргумента

Тип данных параметра функции.

новое_имя

Новое имя семейства операторов.

новый_владелец

Новый владелец семейства операторов.

новая_схема

Новая схема семейства операторов.

Предложения OPERATOR и FUNCTION могут указываться в любом порядке.

Замечания

Заметьте, что в синтаксисе DROP указывается только «слот» в семействе операторов, по номеру стратегии или опорной функции, и входные типы данных. Имя оператора или функции, занимающих этот слот, не упоминается. Также учтите, что в DROP FUNCTION указываются типы входных данных, которые должна поддерживать функция, но для индексов GiST, SP-GiST и GIN они могут не иметь ничего общего с типами фактических аргументов функции.

Так как механизмы индексов не проверяют права доступа к функциям прежде чем вызывать их, включение функций или операторов в семейство операторов по сути даёт всем право на выполнение их. Обычно это не проблема для таких функций, какие бывают полезны в семействе операторов.

Операторы не должны реализовываться в функциях на языке SQL. SQL-функция вероятнее всего будет встроена в вызывающий запрос, что помешает оптимизатору понять, что этот запрос соответствует индексу.

До PostgreSQL 8.4 предложение OPERATOR могло включать указание RECHECK. Теперь это не поддерживается, так как оператор индекса может быть «неточным» и это определяется на ходу в момент выполнения. Это позволяет эффективно справляться с ситуациями, когда оператор может быть или не быть неточным.

Примеры

Следующий пример добавляет опорные функции и операторы смешанных типов в семейство операторов, уже содержащее классы операторов B-дерева для типов данных int4 и int2.

```
ALTER OPERATOR FAMILY integer_ops USING btree ADD

  -- int4 и int2
  OPERATOR 1 < (int4, int2) ,
  OPERATOR 2 <= (int4, int2) ,
  OPERATOR 3 = (int4, int2) ,
  OPERATOR 4 >= (int4, int2) ,
  OPERATOR 5 > (int4, int2) ,
  FUNCTION 1 btint42cmp(int4, int2) ,

  -- int2 и int4
  OPERATOR 1 < (int2, int4) ,
  OPERATOR 2 <= (int2, int4) ,
  OPERATOR 3 = (int2, int4) ,
  OPERATOR 4 >= (int2, int4) ,
  OPERATOR 5 > (int2, int4) ,
  FUNCTION 1 btint24cmp(int2, int4) ;
```

Удаление этих же элементов:

```
ALTER OPERATOR FAMILY integer_ops USING btree DROP

  -- int4 vs int2
  OPERATOR 1 (int4, int2) ,
  OPERATOR 2 (int4, int2) ,
  OPERATOR 3 (int4, int2) ,
  OPERATOR 4 (int4, int2) ,
  OPERATOR 5 (int4, int2) ,
```

```
    FUNCTION 1 (int4, int2) ,

    -- int2 vs int4
    OPERATOR 1 (int2, int4) ,
    OPERATOR 2 (int2, int4) ,
    OPERATOR 3 (int2, int4) ,
    OPERATOR 4 (int2, int4) ,
    OPERATOR 5 (int2, int4) ,
    FUNCTION 1 (int2, int4) ;
```

Совместимость

Команда `ALTER OPERATOR FAMILY` отсутствует в стандарте SQL.

См. также

CREATE OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS

ALTER POLICY

ALTER POLICY — изменить определение политики защиты на уровне строк

Синтаксис

```
ALTER POLICY имя ON имя_таблицы RENAME TO новое_имя

ALTER POLICY имя ON имя_таблицы
    [ TO { имя_роли | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( выражение_использования ) ]
    [ WITH CHECK ( выражение_проверки ) ]
```

Описание

`ALTER POLICY` изменяет определение существующей политики на уровне строк. Заметьте, что `ALTER POLICY` позволяет изменить только набор ролей, для которых применяется политика, и выражения `USING` и `WITH CHECK`. Чтобы изменить другие свойства политики, например команду, к которой она применяется, а также характеристику разрешительная/ограничительная, политику надо удалить и создать заново.

Использовать `ALTER POLICY` может только владелец таблицы (или представления), к которой применяется эта политика.

Во второй форме `ALTER POLICY` список ролей, *выражение_использования* и *выражение_проверки* заменяются независимо, если они указаны. Когда одно из этих предложений опущено, соответствующая часть политики остаётся неизменной.

Параметры

имя

Имя существующей политики, подлежащей изменению.

имя_таблицы

Имя таблицы (возможно, дополненное схемой), к которой применяется эта политика.

новое_имя

Новое имя политики.

имя_роли

Роль (роли), на которую действует политика. В одной команде можно указать несколько ролей. Чтобы применить политику ко всем ролям, укажите `PUBLIC`.

выражение_использования

Выражение `USING` для политики. За подробностями обратитесь к CREATE POLICY.

выражение_проверки

Выражение `WITH CHECK` для политики. За подробностями обратитесь к CREATE POLICY.

Совместимость

`ALTER POLICY` является расширением Postgres Pro.

См. также

CREATE POLICY, DROP POLICY

ALTER PUBLICATION

ALTER PUBLICATION — изменить определение публикации

Синтаксис

```
ALTER PUBLICATION имя ADD TABLE [ ONLY ] имя_таблицы [ * ] [, ...]
ALTER PUBLICATION имя SET TABLE [ ONLY ] имя_таблицы [ * ] [, ...]
ALTER PUBLICATION имя DROP TABLE [ ONLY ] имя_таблицы [ * ] [, ...]
ALTER PUBLICATION имя SET ( параметр_публикации [= значение] [, ... ] )
ALTER PUBLICATION имя OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }
ALTER PUBLICATION имя RENAME TO новое_имя
```

Описание

Команда `ALTER PUBLICATION` может изменять атрибуты публикации.

Первые три формы управляют вхождением таблиц в публикации. Предложение `SET TABLE` заменяет список таблиц в публикации заданным. Предложения `ADD TABLE` и `DROP TABLE` добавляют и удаляют таблицы в публикации, соответственно. Заметьте, что при добавлении таблиц в публикацию, на которую уже оформлена подписка, необходимо выполнить `ALTER SUBSCRIPTION ... REFRESH PUBLICATION` на стороне подписчика, чтобы это изменение вступило в силу.

Четвёртая форма этой команды, показанная в сводке синтаксиса, может изменять все свойства публикации, заданные в CREATE PUBLICATION. Свойства, которые не упоминаются в этой команде, сохраняют предыдущие значения.

Остальные формы команды меняют владельца и имя публикации.

Для выполнения `ALTER PUBLICATION` необходимо владеть данной публикацией. Чтобы добавить таблицу в публикацию, дополнительно нужно быть владельцем этой таблицы. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право `CREATE` в базе данных. Кроме того, новым владельцем публикации `FOR ALL TABLES` должен быть суперпользователь. Однако суперпользователь может менять владельца публикации вне зависимости от этих ограничений.

Параметры

имя

Имя существующей публикации, определение которой изменяется.

имя_таблицы

Имя существующей таблицы. Если перед именем таблицы указано `ONLY`, затрагивается только заданная таблица. Без `ONLY` затрагивается и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно добавить необязательное указание `*`, чтобы явно обозначить, что должны затрагиваться и все дочерние таблицы.

`SET ( параметр_публикации [= значение] [, ... ] )`

Это предложение изменяет параметры публикации, изначально установленные командой CREATE PUBLICATION. За дополнительными сведениями обратитесь к её описанию.

новый_владелец

Имя пользователя, назначаемого новым владельцем публикации.

новое_имя

Новое имя публикации.

Примеры

Изменение публикации, чтобы публиковались только удаления и изменения:

```
ALTER PUBLICATION noinsert SET (publish = 'update, delete');
```

Добавление таблиц в публикацию:

```
ALTER PUBLICATION mypublication ADD TABLE users, departments;
```

Совместимость

`ALTER PUBLICATION` является расширением Postgres Pro.

См. также

CREATE PUBLICATION, DROP PUBLICATION, CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

ALTER ROLE

ALTER ROLE — изменить роль в базе данных

Синтаксис

```
ALTER ROLE указание_роли [ WITH ] параметр [ ... ]

Здесь параметр:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT предел_подключений
    | [ ENCRYPTED ] PASSWORD 'пароль'
    | VALID UNTIL 'дата_время'

ALTER ROLE имя RENAME TO новое_имя

ALTER ROLE { указание_роли | ALL } [ IN DATABASE имя_бд ] SET параметр_конфигурации
 { TO | = } { значение | DEFAULT }
ALTER ROLE { указание_роли | ALL } [ IN DATABASE имя_бд ] SET параметр_конфигурации
 FROM CURRENT
ALTER ROLE { указание_роли | ALL } [ IN DATABASE имя_бд ] RESET параметр_конфигурации
ALTER ROLE { указание_роли | ALL } [ IN DATABASE имя_бд ] RESET ALL

Здесь указание_роли:

    имя_роли
  | CURRENT_USER
  | SESSION_USER
```

Описание

`ALTER ROLE` изменяет атрибуты роли Postgres Pro.

Первая форма команды в этой справке может изменить многие атрибуты роли, которые можно указать в CREATE ROLE. (Покрываются все возможные атрибуты, отсутствуют только возможности добавления/удаления членов роли; для этого нужно использовать GRANT и REVOKE.) Атрибуты, не упомянутые в команде, сохраняют свои предыдущие значения. Суперпользователи базы данных могут изменить любые параметры любой роли, а пользователи с правом `CREATEROLE` могут также менять любые параметры (за исключением параметров `SUPERUSER`, `REPLICATION` и `BYPASSRLS`), но только не ролей суперпользователей и репликации. Обычные пользователи (роли) могут менять только свой пароль.

Вторая форма меняет имя роли. Суперпользователи базы данных могут переименовать любую роль, а пользователи с правом `CREATEROLE` могут переименовывать роли не суперпользователей. Также нельзя переименовать роль текущего пользователя в активном сеансе. (Если вам нужно сделать это, подключитесь другим пользователем.) Так как в паролях с `MD5`-шифрованием имя роли используется в качестве криптосоли, при переименовании роли её пароль очищается, если он был зашифрован `MD5`.

Оставшиеся формы меняют значение по умолчанию конфигурационной переменной, которое будет распространяться на сеансы роли во всех базах данных, либо, если добавлено предложение

IN DATABASE, только на сеансы роли в заданной базе. Если вместо имени роли указано ALL, это значение переменной распространяется на все роли. Использование ALL с IN DATABASE по сути равносильно использованию команды ALTER DATABASE ... SET ....

Когда эта роль впоследствии установит новое подключение, указанное значение станет значением по умолчанию в сеансе, переопределяя значение, заданное в postgresql.conf или полученное из командной строки postgres. Это происходит только в момент входа; при выполнении SET ROLE или SET SESSION AUTHORIZATION новые значения не применятся. Набор параметров для всех баз данных переопределяется параметрами уровня БД, установленными для роли. Параметры для конкретной базы данных или конкретной роли переопределяют параметры для всех ролей.

Суперпользователи могут менять значения переменных по умолчанию для любых ролей, а пользователи с правом CREATEROLE могут менять их только для ролей не суперпользователей. Обычные пользователи могут определять переменные только для себя. Некоторые переменные конфигурации нельзя задать таким способом, а некоторые может настроить только суперпользователь. Параметры всех ролей во всех базах данных могут настраивать только суперпользователи.

Параметры

имя

Имя роли, атрибуты которой изменяются.

CURRENT_USER

Выбирает для изменения текущего пользователя, а не явно задаваемую роль.

SESSION_USER

Выбирает для изменения текущего пользователя сеанса, а не явно задаваемую роль.

SUPERUSER
NOSUPERUSER
CREATEDB
NOCREATEDB
CREATEROLE
NOCREATEROLE
INHERIT
NOINHERIT
LOGIN
NOLOGIN
REPLICATION
NOREPLICATION
BYPASSRLS
NOBYPASSRLS
CONNECTION LIMIT *предел_подключений*
[ENCRYPTED] PASSWORD *пароль*
VALID UNTIL '*дата_время*'

Эти предложения меняют атрибуты, изначально установленные командой CREATE ROLE. За дополнительными сведениями обратитесь к странице справки CREATE ROLE.

новое_имя

Новое имя роли.

имя_бд

Имя базы данных, в которой устанавливается конфигурационная переменная.

параметр_конфигурации
значение

Указанный параметр конфигурации принимает заданное значение по умолчанию в сеансах роли. Если *значение* задано как `DEFAULT` или, что то же самое, применяется операция `RESET`, переопределение этого параметра для роли удаляется и роль будет получать в новых сеансах системное значение параметра. Для очистки значений всех параметров, связанных с ролью, применяется `RESET ALL`. `SET FROM CURRENT` сохраняет текущее значение параметра в активном сеансе в качестве значения для данной роли. Если указано `IN DATABASE`, параметр конфигурации настраивается или удаляется только для данной роли и указанной базы данных.

Определения переменных для роли применяются только в начале сеанса; команды SET ROLE и SET SESSION AUTHORIZATION эти определения не обрабатывают.

За подробными сведениями об именах и значениях параметров обратитесь к SET и Главе 18.

Замечания

Для добавления новых ролей используйте команду CREATE ROLE, а для удаления роли — DROP ROLE.

`ALTER ROLE` не может управлять членством роли, для этого применяется GRANT и REVOKE.

Указывая в этой команде незашифрованный пароль, следует проявлять осторожность. Пароль будет передаваться на сервер открытым текстом и может также записаться в историю команд клиента или в протокол работы сервера. В psql есть команда `\password`, с помощью которой можно сменить пароль роли, не рискуя рассекретить пароль.

Также возможно связать сеансовые значения по умолчанию с определённой базой данных, а не с ролью (см. ALTER DATABASE). В случае конфликта параметры для базы данных и роли переопределяют параметры только для роли, которые, в свою очередь, переопределяют параметры для базы данных.

Примеры

Изменение пароля роли:

```
ALTER ROLE davide WITH PASSWORD 'hu8jmn3';
```

Удаление пароля роли:

```
ALTER ROLE davide WITH PASSWORD NULL;
```

Изменение срока действия пароля (в частности, определяется, что пароль должен перестать действовать в полдень 4 мая 2015 г. в часовом поясе UTC+1):

```
ALTER ROLE chris VALID UNTIL 'May 4 12:00:00 2015 +1';
```

Установка бесконечного срока действия пароля:

```
ALTER ROLE fred VALID UNTIL 'infinity';
```

Наделение роли правами на создание других ролей и новых баз данных:

```
ALTER ROLE miriam CREATEROLE CREATEDB;
```

Определение нестандартного значения параметра maintenance_work_mem для роли:

```
ALTER ROLE worker_bee SET maintenance_work_mem = 100000;
```

Определение нестандартного значения параметра client_min_messages для роли и заданной базы:

```
ALTER ROLE fred IN DATABASE devel SET client_min_messages = DEBUG;
```

Совместимость

Оператор `ALTER ROLE` является расширением Postgres Pro.

См. также

CREATE ROLE, DROP ROLE, ALTER DATABASE, SET

ALTER RULE

ALTER RULE — изменить определение правила

Синтаксис

```
ALTER RULE имя ON имя_таблицы RENAME TO новое_имя
```

Описание

`ALTER RULE` изменяет свойства существующего правила. В настоящее время эта команда может только изменить имя правила.

Использовать `ALTER RULE` может только владелец таблицы (или представления), к которой применяется это правило.

Параметры

имя

Имя существующего правила, подлежащего изменению.

имя_таблицы

Имя (возможно, дополненное схемой) существующей таблицы (или представления), к которой применяется это правило.

новое_имя

Новое имя правила.

Примеры

Переименование существующего правила:

```
ALTER RULE notify_all ON emp RENAME TO notify_me;
```

Совместимость

Оператор `ALTER RULE` является языковым расширением Postgres Pro, как и вся система перезаписи запросов.

См. также

CREATE RULE, DROP RULE

ALTER SCHEMA

ALTER SCHEMA — изменить определение схемы

Синтаксис

```
ALTER SCHEMA имя RENAME TO новое_имя
ALTER SCHEMA имя OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }
```

Описание

`ALTER SCHEMA` изменяет определение схемы.

Выполнить `ALTER SCHEMA` может только владелец соответствующей схемы. Чтобы переименовать схему, необходимо также иметь право `CREATE` в базе данных схемы. Чтобы сменить владельца необходимо быть непосредственным или опосредованным членом новой роли-владельца и иметь право `CREATE` в базе данных. (Суперпользователи наделяются этими правами автоматически.)

Параметры

имя

Имя существующей схемы.

новое_имя

Новое имя схемы. Новое имя не может начинаться с `pg_`, так как такие имена зарезервированы для системных схем.

новый_владелец

Новый владелец схемы.

Совместимость

Оператор `ALTER SCHEMA` отсутствует в стандарте SQL.

См. также

CREATE SCHEMA, DROP SCHEMA

ALTER SEQUENCE

ALTER SEQUENCE — изменить определение генератора последовательности

Синтаксис

```
ALTER SEQUENCE [ IF EXISTS ] имя
    [ AS тип_данных ]
    [ INCREMENT [ BY ] шаг ]
    [ MINVALUE мин_значение | NO MINVALUE ] [ MAXVALUE макс_значение | NO MAXVALUE ]
    [ START [ WITH ] начало ]
    [ RESTART [ [ WITH ] перезапуск ] ]
    [ CACHE кеш ] [ [ NO ] CYCLE ]
    [ OWNED BY { имя_таблицы.имя_столбца | NONE } ]
ALTER SEQUENCE [ IF EXISTS ] имя OWNER TO { новый_владелец | CURRENT_USER |
 SESSION_USER }
ALTER SEQUENCE [ IF EXISTS ] имя RENAME TO новое_имя
ALTER SEQUENCE [ IF EXISTS ] имя SET SCHEMA новая_схема
```

Описание

`ALTER SEQUENCE` меняет параметры существующего генератора последовательности. Параметры, не определяемые явно в команде `ALTER SEQUENCE`, сохраняют свои предыдущие значения.

Выполнить `ALTER SEQUENCE` может только владелец соответствующей последовательности. Чтобы сменить схему последовательности, необходимо также иметь право `CREATE` в новой схеме. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме последовательности. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать последовательность. Однако суперпользователь может сменить владельца последовательности в любом случае.)

Параметры

имя

Имя (возможно, дополненное схемой) последовательности, подлежащей изменению.

`IF EXISTS`

Не считать ошибкой, если последовательность не существует. В этом случае будет выдано замечание.

тип_данных

Необязательное предложение `AS` *тип_данных* меняет тип данных для последовательности. Допустимые типы: `smallint`, `integer` и `bigint`.

При изменении типа данных автоматически меняются минимальное и максимальное значения последовательности, в том и только в том случае, если это были минимальные и максимальные значения старого типа данных (другими словами, если последовательность была создана со свойствами `NO MINVALUE` или `NO MAXVALUE`, неявно или явно). В противном случае минимальные и максимальные значения сохраняются, если только в этой же команде не указаны новые значения. Если минимальное/максимальное значение не умещается в новом типе данных, выдаётся ошибка.

шаг

Предложение `INCREMENT BY` *шаг* является необязательным. При положительном значении шага генерируется возрастающая последовательность, при отрицательном — убывающая; если шаг не указан, сохраняется предыдущее значение.

мин_значение
NO MINVALUE

Необязательное предложение MINVALUE *мин_значение* определяет минимальное значение, которое будет генерировать данная последовательность. Если указано NO MINVALUE, для возрастающей последовательности этим значением будет 1, а для убывающей — минимальное число для её типа данных. В отсутствие этих указаний будет сохранено текущее минимальное значение.

макс_значение
NO MAXVALUE

Необязательное предложение MAXVALUE *макс_значение* определяет максимальное значение, которое будет генерировать данная последовательность. Если указано NO MAXVALUE, для возрастающей последовательности этим значением будет максимальное число для её типа данных, а для убывающей — -1. В отсутствие этих указаний будет сохранено текущее максимальное значение.

начало

Необязательное предложение START WITH *начало* меняет записанное начальное значение последовательности. При этом *текущее* значение последовательности не меняется, а только устанавливается значение, которое будет применено будущими командами ALTER SEQUENCE RESTART.

перезапуск

Необязательное предложение RESTART [WITH *перезапуск*] меняет текущее значение последовательности. Оно подобно вызову функции setval с параметром is_called = false: указанное значение перезапуска будет возвращено при *следующем* вызове функции nextval. Отсутствие в RESTART значения *перезапуск* равносильно передаче стартового значения, записанного командой CREATE SEQUENCE или последнего установленного командой ALTER SEQUENCE START WITH.

В отличие от вызова setval, операция RESTART с последовательностью является транзакционной и не даёт параллельным транзакциям получать числа из той же последовательности. Если это поведение не устраивает, следует воспользоваться функцией setval.

кеш

Предложение CACHE *кеш* разрешает предварительно выделять и сохранять в памяти числа последовательности для ускорения доступа к ним. Минимальное значение равно 1 (т. е. за один раз генерируется только одно значение, кеширования нет). Если это предложение отсутствует, сохраняется старое значение размера кеша.

CYCLE

Необязательное ключевое слово CYCLE позволяет зациклить последовательность при достижении *макс_значения* или *мин_значения* для возрастающей и убывающей последовательности, соответственно. Когда этот предел достигается, следующим числом этих последовательностей будет соответственно *мин_значение* или *макс_значение*.

NO CYCLE

Если добавляется необязательное указание NO CYCLE, при каждом вызове nextval после достижения предельного значения будет возникать ошибка. Если же указания CYCLE и NO CYCLE отсутствуют, сохраняется предыдущее поведение зацикливания.

OWNED BY *имя_таблицы.имя_столбца*
OWNED BY NONE

Указание OWNED BY связывает последовательность с определённым столбцом таблицы, с тем чтобы при удалении этого столбца (или всей таблицы) автоматически удалилась и

последовательность. Это указание заменяет любую ранее установленную связь данной последовательности. Целевая таблица должна иметь того же владельца и находиться в той же схеме, что и последовательность. Указание `OWNED BY NONE` убирает все существующие связи, обозначая последовательность «независимой».

`новый_владелец`

Имя пользователя, назначаемого новым владельцем последовательности.

`новое_имя`

Новое имя последовательности.

`новая_схема`

Новая схема последовательности.

Замечания

`ALTER SEQUENCE` не оказывает немедленного влияния на результаты `nextval` в серверных процессах, кроме текущего, которые могли предварительно сгенерировать (кешировать) значения последовательности. Эти процессы заметят изменения только после того, как будут израсходованы все кешированные значения. Текущий серверный процесс реагирует на изменения сразу.

`ALTER SEQUENCE` не влияет на значение `currval` последовательности. (В PostgreSQL до версии 8.3 это могло происходить.)

`ALTER SEQUENCE` блокирует параллельные вызовы `nextval`, `currval`, `lastval` и `setval`.

По историческим причинам `ALTER TABLE` тоже может работать с последовательностями, но все разновидности `ALTER TABLE`, допустимые для управления последовательностями, равнозначны вышеперечисленным формам.

Примеры

Перезапуск последовательности `serial` с числа 105:

```
ALTER SEQUENCE serial RESTART WITH 105;
```

Совместимость

Оператор `ALTER SEQUENCE` соответствует стандарту SQL, за исключением предложений `AS`, `START WITH`, `OWNED BY`, `OWNER TO`, `RENAME TO` и `SET SCHEMA`, являющихся расширениями Postgres Pro.

См. также

CREATE SEQUENCE, DROP SEQUENCE

ALTER SERVER

ALTER SERVER — изменить определение стороннего сервера

Синтаксис

```
ALTER SERVER имя [ VERSION 'новая_версия' ]
    [ OPTIONS ( [ ADD | SET | DROP ] параметр ['значение'] [, ... ] ) ]
ALTER SERVER имя OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }
ALTER SERVER имя RENAME TO новое_имя
```

Описание

`ALTER SERVER` изменяет определение стороннего сервера. Первая форма меняет строку версии сервера или общие параметры сервера (требуется минимум одно предложение). Вторая форма меняет владельца сервера.

Изменить свойства сервера может только его владелец. Чтобы изменить владельца, необходимо быть его владельцем, а также непосредственным или опосредованным членом новой роли-владельца, и кроме того, иметь право `USAGE` для обёртки сторонних данных сервера. (Суперпользователи удовлетворяют всем этим условиям автоматически.)

Параметры

имя

Имя существующего сервера.

новая_версия

Новая версия сервера.

`OPTIONS ( [ ADD | SET | DROP ] параметр ['значение'] [, ... ] )`

Эти формы изменяют параметры сервера. Указания `ADD`, `SET` и `DROP` определяют выполняемое действие (добавление, установка и удаление, соответственно). Если действие не задано явно, подразумевается `ADD`. Имена параметров должны быть уникальными, они вместе со значениями также проверяются библиотекой обёртки сторонних данных.

новый_владелец

Имя пользователя, назначаемого новым владельцем стороннего сервера.

новое_имя

Новое имя стороннего сервера.

Примеры

Изменение свойств сервера `foo`, добавление параметров подключения:

```
ALTER SERVER foo OPTIONS (host 'foo', dbname 'foodb');
```

Изменение свойств сервера `foo`: смена версии, изменение параметра `host`:

```
ALTER SERVER foo VERSION '8.4' OPTIONS (SET host 'baz');
```

Совместимость

`ALTER SERVER` соответствует стандарту ISO/IEC 9075-9 (SQL/MED). Формы `OWNER TO` и `RENAME` являются расширениями Postgres Pro.

См. также

CREATE SERVER, DROP SERVER

ALTER STATISTICS

ALTER STATISTICS — изменить определение объекта расширенной статистики

Синтаксис

```
ALTER STATISTICS имя OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }
ALTER STATISTICS имя RENAME TO новое_имя
ALTER STATISTICS имя SET SCHEMA новая_схема
```

Описание

`ALTER STATISTICS` меняет параметры существующего объекта расширенной статистики. Параметры, не определённые явно в команде `ALTER STATISTICS`, сохраняют свои предыдущие значения.

Выполнить `ALTER STATISTICS` может только владелец объекта статистики. Чтобы сменить схему объекта статистики, необходимо также иметь право `CREATE` в новой схеме. Чтобы сменить владельца, также нужно быть непосредственным или опосредованным членом новой роли-владельца, и эта роль должна иметь право `CREATE` в схеме объекта статистики. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать объект статистики. Однако суперпользователь может сменить владельца объекта статистики в любом случае.)

Параметры

имя

Имя (возможно, дополненное схемой) объекта статистики, подлежащего изменению.

новый_владелец

Имя пользователя, назначаемого новым владельцем объекта статистики.

новое_имя

Новое имя объекта статистики.

новая_схема

Новая схема объекта статистики.

Совместимость

Оператор `ALTER STATISTICS` отсутствует в стандарте SQL.

См. также

CREATE STATISTICS, DROP STATISTICS

ALTER SUBSCRIPTION

ALTER SUBSCRIPTION — изменить определение подписки

Синтаксис

```
ALTER SUBSCRIPTION имя CONNECTION 'строка_подключения'
ALTER SUBSCRIPTION имя SET PUBLICATION имя_публикации [, ...] [ WITH
 ( параметр_set_publication [= значение] [, ... ] ) ]
ALTER SUBSCRIPTION имя REFRESH PUBLICATION [ WITH ( параметр_обновления [= значение]
 [, ... ] ) ]
ALTER SUBSCRIPTION имя ENABLE
ALTER SUBSCRIPTION имя DISABLE
ALTER SUBSCRIPTION имя SET ( параметр_подписки [= значение] [, ... ] )
ALTER SUBSCRIPTION имя OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }
ALTER SUBSCRIPTION имя RENAME TO новое_имя
```

Описание

`ALTER SUBSCRIPTION` может менять многие свойства подписки, которые могут задаваться в CREATE SUBSCRIPTION.

Чтобы выполнить `ALTER SUBSCRIPTION` для подписки, нужно быть её владельцем. Чтобы сменить владельца, нужно быть непосредственным или опосредованным членом новой роли-владельца. Новый владелец должен быть суперпользователем. (В настоящее время все владельцы подписок должны быть суперпользователями, так что на практике проверка владельца будет пропущена. Но в будущем это может быть изменено.)

Параметры

имя

Имя подписки, свойства которой изменяются.

`CONNECTION '`*строка_подключения*`'`

Это предложение изменяет строку соединения, изначально установленную командой CREATE SUBSCRIPTION. За дополнительными сведениями обратитесь к описанию этой команды.

`SET PUBLICATION` *имя_публикации*

Изменяет список публикаций, на которые оформлена подписка. За подробностями обратитесь к описанию CREATE SUBSCRIPTION. По умолчанию эта команда также выполняет действие `REFRESH PUBLICATION`.

В указании *параметр_set_publication* задаются дополнительные свойства операции. Поддерживаются следующие параметры:

`refresh` (`boolean`)

Со значением `false` данная команда не будет обновлять информацию о таблицах. В этом случае следует выполнить `REFRESH PUBLICATION` отдельно. Значение по умолчанию — `true`.

Кроме того, здесь могут задаваться параметры обновления, упомянутые в описании `REFRESH PUBLICATION`.

`REFRESH PUBLICATION`

Считывает недостающую информацию о таблицах с публикующего сервера. В результате производится репликация таблиц, добавленных в публикации, на которые оформлена подписка, после последнего вызова `REFRESH PUBLICATION` или `CREATE SUBSCRIPTION`.

В указании *параметр_обновления* задаются дополнительные свойства операции обновления. Поддерживаются следующие параметры:

`copy_data` (`boolean`)

Определяет, должны ли копироваться существующие данные в публикациях, на которые оформляется подписка, сразу после начала репликации. Значение по умолчанию — `true`. (К таблицам, добавленным в публикации ранее, это не относится.)

`ENABLE`

Включает ранее отключённую подписку, запуская процесс логической репликации в конце транзакции.

`DISABLE`

Отключает активную подписку, останавливая процесс логической репликации в конце транзакции.

`SET ( *параметр_подписки* [= *значение*] [, ... ] )`

Это предложение изменяет параметры, изначально установленные командой CREATE SUBSCRIPTION. За подробностями обратитесь к её описанию. Данное предложение позволяет изменить параметры `slot_name` и `synchronous_commit`

новый_владелец

Имя пользователя, назначаемого новым владельцем подписки.

новое_имя

Новое имя подписки.

Примеры

Изменение подписки, заключающееся в подписывании на публикацию `insert_only`:

```
ALTER SUBSCRIPTION mysub SET PUBLICATION insert_only;
```

Отключение (остановка) подписки:

```
ALTER SUBSCRIPTION mysub DISABLE;
```

Совместимость

`ALTER SUBSCRIPTION` является расширением Postgres Pro.

См. также

CREATE SUBSCRIPTION, DROP SUBSCRIPTION, CREATE PUBLICATION, ALTER PUBLICATION

ALTER SYSTEM

ALTER SYSTEM — изменить параметр конфигурации сервера

Синтаксис

```
ALTER SYSTEM SET параметр_конфигурации { TO | = } { значение | 'значение' | DEFAULT }

ALTER SYSTEM RESET параметр_конфигурации
ALTER SYSTEM RESET ALL
```

Описание

Оператор `ALTER SYSTEM` применяется для изменения параметров конфигурации сервера, распространяющихся на весь кластер баз данных. Пользоваться им может быть удобнее, чем вручную редактировать файл `postgresql.conf`. `ALTER SYSTEM` записывает заданное значение параметра в файл `postgresql.auto.conf`, который считывается сервером в дополнение к `postgresql.conf`. При указании в качестве значения параметра `DEFAULT` или применении формы `RESET` соответствующий элемент конфигурации удаляется из `postgresql.auto.conf`. Удалить все настроенные таким способом параметры позволяет предложение `RESET ALL`.

Значения, установленные командой `ALTER SYSTEM`, вступают в силу после следующей перезагрузки конфигурации сервера либо после перезапуска сервера (если это параметры, воспринимаемые только при запуске). Перезагрузить конфигурацию сервера можно, вызвав SQL-функцию `pg_reload_conf()`, выполнив `pg_ctl reload` или отправив сигнал SIGHUP главному серверному процессу.

Выполнить `ALTER SYSTEM` могут только суперпользователи. А так как эта команда работает непосредственно с файловой системой и не может быть отменена, её нельзя поместить в блок транзакции или функцию.

Параметры

параметр_конфигурации

Имя устанавливаемого параметра конфигурации. Список доступных параметров приведён в Главе 18.

значение

Новое значение параметра. Значениями могут быть строковые константы, идентификаторы, числа или списки таких элементов через запятую, в зависимости от конкретного параметра. Если в качестве значения указать `DEFAULT`, параметр и его значение удаляется из `postgresql.auto.conf`.

Замечания

С помощью этой команды нельзя задать data_directory, равно как и другие параметры, недопустимые в `postgresql.conf` (например, предустановленные параметры).

Другие способы настройки параметров описаны в Разделе 18.1.

Примеры

Установка уровня ведения журнала транзакций (`wal_level`):

```
ALTER SYSTEM SET wal_level = replica;
```

Отмена изменения, восстановление значения, заданного в `postgresql.conf`:

```
ALTER SYSTEM RESET wal_level;
```

Совместимость

Оператор `ALTER SYSTEM` является расширением Postgres Pro.

См. также

SET, SHOW

ALTER TABLE

ALTER TABLE — изменить определение таблицы

Синтаксис

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] имя [ * ]
    действие [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] имя [ * ]
    RENAME [ COLUMN ] имя_столбца TO новое_имя_столбца
ALTER TABLE [ IF EXISTS ] [ ONLY ] имя [ * ]
    RENAME CONSTRAINT имя_ограничения TO имя_нового_ограничения
ALTER TABLE [ IF EXISTS ] имя
    RENAME TO новое_имя
ALTER TABLE [ IF EXISTS ] имя
    SET SCHEMA новая_схема
ALTER TABLE ALL IN TABLESPACE имя [ OWNED BY имя_роли [, ... ] ]
    SET TABLESPACE новое_табл_пространство [ NOWAIT ]
ALTER TABLE [ IF EXISTS ] имя
    ATTACH PARTITION имя_секции FOR VALUES указание_границ_секции
ALTER TABLE [ IF EXISTS ] имя
    DETACH PARTITION имя_секции

Где действие может быть следующим:

    ADD [ COLUMN ] [ IF NOT EXISTS ] имя_столбца тип_данных
 [ COLLATE правило_сортировки ] [ ограничение_столбца [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] имя_столбца [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] имя_столбца [ SET DATA ] TYPE тип_данных
 [ COLLATE правило_сортировки ] [ USING выражение ]
    ALTER [ COLUMN ] имя_столбца SET DEFAULT выражение
    ALTER [ COLUMN ] имя_столбца DROP DEFAULT
    ALTER [ COLUMN ] имя_столбца { SET | DROP } NOT NULL
    ALTER [ COLUMN ] имя_столбца ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
 [ ( параметры_последовательности ) ]
    ALTER [ COLUMN ] имя_столбца { SET GENERATED { ALWAYS | BY DEFAULT } |
 SET параметр_последовательности | RESTART [ [ WITH ] перезапуск ] } [...]
    ALTER [ COLUMN ] имя_столбца DROP IDENTITY [ IF EXISTS ]
    ALTER [ COLUMN ] имя_столбца SET STATISTICS integer
    ALTER [ COLUMN ] имя_столбца SET ( атрибут = значение [, ... ] )
    ALTER [ COLUMN ] имя_столбца RESET ( атрибут [, ... ] )
    ALTER [ COLUMN ] имя_столбца SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
    ADD ограничение_таблицы [ NOT VALID ]
    ADD ограничение_таблицы_по_индексу
    ALTER CONSTRAINT имя_ограничения [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY
 DEFERRED | INITIALLY IMMEDIATE ]
    VALIDATE CONSTRAINT имя_ограничения
    DROP CONSTRAINT [ IF EXISTS ]  имя_ограничения [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ имя_триггера | ALL | USER ]
    ENABLE TRIGGER [ имя_триггера | ALL | USER ]
    ENABLE REPLICA TRIGGER имя_триггера
    ENABLE ALWAYS TRIGGER имя_триггера
    DISABLE RULE имя_правила_перезаписи
    ENABLE RULE имя_правила_перезаписи
    ENABLE REPLICA RULE имя_правила_перезаписи
    ENABLE ALWAYS RULE имя_правила_перезаписи
```

```
    DISABLE ROW LEVEL SECURITY
    ENABLE ROW LEVEL SECURITY
    FORCE ROW LEVEL SECURITY
    NO FORCE ROW LEVEL SECURITY
    CLUSTER ON имя_индекса
    SET WITHOUT CLUSTER
    SET WITH OIDS
    SET WITHOUT OIDS
    SET TABLESPACE новое_табл_пространство
    SET { LOGGED | UNLOGGED }
    SET ( параметр_хранения [= значение] [, ... ] )
    RESET ( параметр_хранения [, ... ] )
    INHERIT таблица_родитель
    NO INHERIT таблица_родитель
    OF имя_типа
    NOT OF
    OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }
    REPLICA IDENTITY { DEFAULT | USING INDEX имя_индекса | FULL | NOTHING }
```

и *ограничение_таблицы_по_индексу*:

```
    [ CONSTRAINT имя_ограничения ]
    { UNIQUE | PRIMARY KEY } USING INDEX имя_индекса
    [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

Описание

`ALTER TABLE` меняет определение существующей таблицы. Несколько её разновидностей описаны ниже. Заметьте, что для разных разновидностей могут требоваться разные уровни блокировок. Если явно не отмечено другое, запрашивается блокировка `ACCESS EXCLUSIVE`. При указании нескольких подкоманд будет запрашиваться самая сильная блокировка из требуемых ими.

`ADD COLUMN [ IF NOT EXISTS ]`

Эта форма добавляет в таблицу новый столбец, с тем же синтаксисом, что и CREATE TABLE. Если указано `IF NOT EXISTS` и столбец с таким именем уже существует, это не будет ошибкой.

`DROP COLUMN [ IF EXISTS ]`

Эта форма удаляет столбец из таблицы. При этом автоматически будут удалены индексы и ограничения таблицы, связанные с этим столбцом. Также будет удалена многовариантная статистика, охватывающая удаляемый столбец, если после его удаления в статистике останутся данные только одного столбца. Если от этого столбца зависят какие либо объекты вне этой таблицы, например, внешние ключи или представления, чтобы удалить их, необходимо добавить указание `CASCADE`. Если в команде указано `IF EXISTS` и этот столбец не существует, это не считается ошибкой, вместо этого просто выдаётся замечание.

`SET DATA TYPE`

Эта форма меняет тип столбца таблицы. Индексы и простые табличные ограничения, включающие этот столбец, будут автоматически преобразованы для использования нового типа столбца, для чего будет заново разобрано определяющее их выражение. Необязательное предложение `COLLATE` задаёт правило сортировки для нового столбца; если оно опущено, выбирается правило сортировки по умолчанию для нового типа. Необязательное предложение `USING` определяет, как новое значение столбца будет получено из старого; если оно отсутствует, выполняется приведение типа по умолчанию, как обычное присваивание значения старого типа новому. Предложение `USING` становится обязательным, если неявное приведение или присваивание с приведением старого типа к новому не определено.

SET/DROP DEFAULT

Эти формы задают или удаляют значение по умолчанию для столбцов. Значения по умолчанию применяются только при последующих командах INSERT или UPDATE; их изменения не отражаются в строках, уже существующих в таблице.

SET/DROP NOT NULL

Эти формы определяют, будет ли столбец принимать значения NULL или нет. Задать SET NOT NULL можно, только если столбец не содержит значений NULL.

Если данная таблица является секцией, операцию DROP NOT NULL нельзя выполнить для столбца, если он определён с характеристикой NOT NULL в родительской таблице. Чтобы удалить ограничение NOT NULL из всех секций, выполните DROP NOT NULL для родительской таблицы. Даже если ограничение NOT NULL в родительской таблице отсутствует, при желании такое ограничение может быть добавлено в отдельные секции. Другими словами, потомки могут запрещать значения NULL, даже если родитель их допускает, но не наоборот.

```
ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
SET GENERATED { ALWAYS | BY DEFAULT }
DROP IDENTITY [ IF EXISTS ]
```

Эти формы определяют, является ли столбец столбцом идентификации, и меняют свойства генерирования значений уже существующего столбца идентификации. За подробностями обратитесь к CREATE TABLE.

Если указано DROP IDENTITY IF EXISTS и заданный столбец не является столбцом идентификации, это не считается ошибкой. В этом случае выдаётся только замечание.

```
SET параметр_последовательности
RESTART
```

Эти формы меняют нижележащую последовательность ранее созданного столбца идентификации. Здесь *параметр_последовательности* — это параметр, поддерживаемый командой ALTER SEQUENCE, например INCREMENT BY.

SET STATISTICS

Эта форма задаёт ориентир сбора статистики по столбцу для последующих операций ANALYZE. Диапазон допустимых значений ориентира: 0..10000; при -1 применяется системное значение по умолчанию (default_statistics_target). За дополнительными сведениями об использовании статистики планировщиком запросов Postgres Pro обратитесь к Разделу 14.2.

SET STATISTICS запрашивает блокировку SHARE UPDATE EXCLUSIVE.

```
SET ( атрибут = значение [, ... ] )
RESET ( атрибут [, ... ] )
```

Эта форма устанавливает или сбрасывает параметры атрибутов. В настоящее время единственными параметрами атрибутов являются n_distinct и n_distinct_inherited, которые переопределяют оценку кол-ва_различных_значений, производимую последующими операциями ANALYZE. Атрибут n_distinct влияет на расчёт статистики по самой таблице, а n_distinct_inherited — на статистику по таблице и её потомкам. Если заданное значение положительно, ANALYZE будет считать, что столбец содержит именно это количество различных значений не NULL. Если заданное значение отрицательно (оно должно быть больше или равно -1), ANALYZE будет считать, что количество различных значений не NULL в столбце линейно зависит от размера таблицы; точное число будет получено умножением примерного размера таблицы на абсолютное значение параметра. Например, при -1 будет предполагаться, что различны все значения в столбце, а при -0,5 — что в среднем каждое значение повторяется дважды. Это может быть полезно, когда размер таблицы меняется со временем, так как умножение на число строк в таблице производится только во время планирования запроса. С 0

количество различных значений оценивается как обычно. За дополнительными сведениями об использовании статистики планировщиком запросов Postgres Pro обратитесь к Разделу 14.2.

Для изменения параметров атрибутов запрашивается блокировка SHARE UPDATE EXCLUSIVE.

SET STORAGE

Эта форма устанавливает режим хранения столбца. Она определяет, хранятся ли данные внутри таблицы или в отдельной таблице TOAST, а также, сжимаются ли они. Режим PLAIN должен применяться для значений фиксированной длины, таких как integer; это вариант хранения внутри, без сжатия. Режим MAIN применяется для хранения внутри, но сжатых данных, EXTERNAL — для внешнего хранения несжатых данных, а EXTENDED — для внешнего хранения сжатых данных. EXTENDED используется по умолчанию для большинства типов данных, поддерживающих хранилище не PLAIN. Применение EXTERNAL позволяет ускорить операции с подстроками на очень больших значениях text и bytea, за счёт проигрыша в объёме хранилища. Заметьте, что предложение SET STORAGE само по себе не меняет ничего в таблице, оно только задаёт стратегию, которая будет реализована при будущих изменениях в таблице. За дополнительными сведениями обратитесь к Разделу 63.2.

ADD *ограничение_таблицы* [NOT VALID]

Эта форма добавляет в таблицу новое ограничение, принимая тот же синтаксис описания ограничения, что и CREATE TABLE, а также дополнительное указание NOT VALID, которое в настоящее время поддерживается только для ограничений внешнего ключа и ограничений-проверок.

Обычно эта форма влечёт сканирование всей таблицы для проверки, что все существующие строки в таблице удовлетворяют новому ограничению. Но если используется указание NOT VALID, эта потенциально длительная проверка пропускается. Тем не менее это ограничение будет действовать при последующих добавлениях или изменениях данных (то есть эти операции не будут выполнены, если новая строка нарушит условие ограничения-проверки, либо при наличии внешнего ключа в главной таблице не найдётся соответствующая строка). Но база данных не будет считать, что ограничение выполняется для всех строк таблицы, пока оно не будет проверено с применением указания VALIDATE CONSTRAINT. Дополнительную информацию об использовании указания NOT VALID вы найдете ниже, в Разделе «Замечания».

Хотя почти все разновидности ADD *ограничение_таблицы* требуют блокировку ACCESS EXCLUSIVE, для указания ADD FOREIGN KEY требуется только блокировка SHARE ROW EXCLUSIVE. Заметьте, что ADD FOREIGN KEY запрашивает такую блокировку как в таблице, в которой добавляется это ограничение, так и в таблице, на которую это ограничение ссылается.

ADD *ограничение_таблицы_по_индексу*

Эта форма добавляет в таблицу новое ограничение PRIMARY KEY или UNIQUE на базе существующего уникального индекса. В это ограничение будут включены все столбцы данного индекса.

Индекс не может быть частичным и включать столбцы-выражения. Кроме того, это должен быть индекс-B-дерево с порядком сортировки по умолчанию. С такими ограничениями добавляемые индексы не будут ничем отличаться от индексов, создаваемых обычными командами ADD PRIMARY KEY и ADD UNIQUE.

В случае с указанием PRIMARY KEY, если столбцы индекса ещё не помечены NOT NULL, данная команда попытается выполнить ALTER COLUMN SET NOT NULL для каждого столбца. При этом потребуется произвести полное сканирование таблицы, чтобы убедиться, что столбец(ы) не содержит NULL. Во всех остальных случаях это быстрая операция.

Если задано имя ограничения, индекс будет переименован и получит заданное имя. В противном случае именем ограничения станет имя индекса.

После выполнения этой команды индекс становится «принадлежащим» ограничению, так же, как если бы он был создан обычной командой `ADD PRIMARY KEY` или `ADD UNIQUE`. Это значит, в частности, что при удалении ограничения индекс будет удалён вместе с ним.

> **Примечание**
>
> Добавление ограничения на базе существующего индекса бывает полезно в ситуациях, когда новое ограничение требуется добавить, не блокируя изменения в таблице на долгое время. Для этого можно создать индекс командой `CREATE INDEX CONCURRENTLY`, а затем задействовать его как полноценное ограничение, используя эту запись. См. следующий пример.

`ALTER CONSTRAINT`

Эта форма меняет атрибуты созданного ранее ограничения. В настоящее время изменять можно только ограничения внешнего ключа.

`VALIDATE CONSTRAINT`

Эта форма проверяет ограничение внешнего ключа или ограничение-проверку, созданное ранее с указанием `NOT VALID`, сканируя всю таблицу с целью убедиться, что ограничению удовлетворяют все строки. Если ограничение уже помечено как проверенное, ничего не происходит. (В чём польза этой команды, вы можете узнать в Разделе «Замечания».)

Эта команда запрашивает блокировку `SHARE UPDATE EXCLUSIVE`.

`DROP CONSTRAINT [ IF EXISTS ]`

Эта форма удаляет указанное ограничение таблицы. Если указано `IF EXISTS` и заданное ограничение не существует, это не считается ошибкой. В этом случае выдаётся только замечание.

`DISABLE/ENABLE [ REPLICA | ALWAYS ] TRIGGER`

Эти формы настраивают срабатывание триггера(ов), принадлежащего таблице. Отключённый триггер сохраняется в системе, но не выполняется, когда происходит вызывающее его событие. Для отложенных триггеров состояние включения проверяется при возникновении события, а не когда фактически вызывается функция триггера. Эта команда может отключить или включить один триггер по имени, либо все триггеры таблицы, либо только пользовательские триггеры (исключая сгенерированные внутрисистемные триггеры ограничений, например, триггеры, реализующие ограничения внешнего ключа или отложенные ограничения уникальности или исключений). Для отключения или включения сгенерированных внутрисистемных триггеров ограничений требуются права суперпользователя; отключать их следует с осторожностью, так как очевидно, что невозможно гарантировать целостность ограничений, если триггеры не работают. На механизм срабатывания триггеров также влияет конфигурационная переменная session_replication_role. Включённые без дополнительных указаний триггеры будут срабатывать, когда роль репликации — «origin» (по умолчанию) или «local». Триггеры, включённые указанием `ENABLE REPLICA`, будут срабатывать, только если текущий режим сеанса — «replica», а после указания `ENABLE ALWAYS` триггеры срабатывают вне зависимости от текущего режима репликации.

Эта команда запрашивает блокировку `SHARE ROW EXCLUSIVE`.

`DISABLE/ENABLE [ REPLICA | ALWAYS ] RULE`

Эти формы настраивают срабатывание правил перезаписи, относящихся к таблице. Отключённое правило сохраняется в системе, но не применяется во время переписывания запроса. По сути эти операции подобны операциям включения/отключения триггеров. Однако это не распространяется на правила `ON SELECT` — они применяются всегда, чтобы

представления продолжали работать, даже в сеансах, исполняющих не основную роль репликации.

`DISABLE/ENABLE ROW LEVEL SECURITY`

Эти формы управляют применением относящихся к таблице политик защиты строк. Если защита включается, но политики для таблицы не определены, применяется политика запрета доступа по умолчанию. Заметьте, что политики могут быть определены для таблицы, даже если защита на уровне строк отключена — в этом случае политики НЕ применяются и их ограничения игнорируются. См. также CREATE POLICY.

`NO FORCE/FORCE ROW LEVEL SECURITY`

Эти формы управляют применением относящихся к таблице политик защиты строк, когда пользователь является её владельцем. Если это поведение включается, политики защиты на уровне строк будут действовать и на владельца таблицы. Если оно отключено (по умолчанию), защита на уровне строк не будет действовать на пользователя, являющегося владельцем таблицы. См. также CREATE POLICY.

`CLUSTER ON`

Эта форма выбирает индекс по умолчанию для последующих операций CLUSTER. Собственно кластеризация таблицы при этом не выполняется.

Для изменения параметров кластеризации запрашивается блокировка `SHARE UPDATE EXCLUSIVE`.

`SET WITHOUT CLUSTER`

Эта форма удаляет последнее заданное указание индекса для CLUSTER. Её действие отразится на будущих операциях кластеризации, для которых не будет задан индекс.

Для изменения параметров кластеризации запрашивается блокировка `SHARE UPDATE EXCLUSIVE`.

`SET WITH OIDS`

Эта форма добавляет в таблицу системный столбец `oid` (см. Раздел 5.4). Если в таблице уже есть такой столбец, она не делает ничего.

Заметьте, что это не равнозначно команде `ADD COLUMN oid oid` (эта команда добавит не системный, а обычный столбец с подходящим именем `oid`).

`SET WITHOUT OIDS`

Эта форма удаляет из таблицы системный столбец `oid`. Это в точности равнозначно `DROP COLUMN oid RESTRICT`, за исключением того, что в случае отсутствия столбца `oid` ошибки не будет.

`SET TABLESPACE`

Эта форма меняет табличное пространство таблицы на заданное и перемещает файлы данных, связанные с таблицей, в новое пространство. Индексы таблицы, если они имеются, не перемещаются; однако их можно переместить отдельно дополнительными командами `SET TABLESPACE`. Форма `ALL IN TABLESPACE` позволяет перенести в другое табличное пространство все таблицы текущей базы данных, при этом она сначала блокирует все таблицы, а затем переносит каждую из них. Эта форма также поддерживает указание `OWNED BY`, с которым перемещаются только таблицы указанного владельца. Если указан параметр `NOWAIT`, команда завершится ошибкой, если не сможет получить все требуемые блокировки немедленно. Заметьте, что системные каталоги эта форма не перемещает; если требуется переместить их, следует использовать `ALTER DATABASE` или явные вызовы `ALTER TABLE`. Отношения `information_schema` не считаются частью системных каталогов и подлежат перемещению. См. также CREATE TABLESPACE.

SET { LOGGED | UNLOGGED }

Эта форма меняет характеристику журналирования таблицы, делает таблицу журналируемой/нежурналируемой, соответственно (см. UNLOGGED). К временной таблице она неприменима.

SET (*параметр_хранения* [= *значение*] [, ...])

Эта форма меняет один или несколько параметров хранения таблицы. Подробнее допустимые параметры описаны в Подразделе «Параметры хранения». Заметьте, что эта команда не меняет содержимое таблицы немедленно; в зависимости от параметра может потребоваться перезаписать таблицы, чтобы получить желаемый эффект. Это можно сделать с помощью команд VACUUM FULL, CLUSTER или одной из форм ALTER TABLE, принудительно перезаписывающих таблицу. Изменения параметров, связанных с планировщиком, вступают в силу при следующей блокировке таблицы, так что в текущих запросах они не проявляются.

Данная форма затребует блокировку SHARE UPDATE EXCLUSIVE для изменения параметров хранения, связанных с фактором заполнения и автоочисткой, а также параметра планировщика parallel_workers.

> **Примечание**
>
> Хотя CREATE TABLE позволяет указать OIDS в синтаксисе WITH (*параметр_хранения*), ALTER TABLE не воспринимает OIDS как параметр хранения. Поэтому для изменения характеристики OID следует применять формы SET WITH OIDS и SET WITHOUT OIDS.

RESET (*параметр_хранения* [, ...])

Эта форма сбрасывает один или несколько параметров хранения к значениям по умолчанию. Как и с SET, для полного обновления таблицы может потребоваться перезаписать таблицу.

INHERIT *таблица_родитель*

Эта форма назначает целевую таблицу потомком заданной родительской таблицы. Впоследствии запросы к родительской таблице будут включать записи и целевой таблицы. Чтобы таблица могла стать потомком, она должна содержать те же столбцы, что и родительская (хотя она может включать и дополнительные столбцы). Столбцы должны иметь одинаковые типы данных и, если в родительской таблице какие-то из них имеют ограничение NOT NULL, они должны иметь ограничение NOT NULL и в таблице-потомке.

Также в таблице-потомке должны присутствовать все ограничения CHECK родительской таблицы, за исключением ненаследуемых (то есть созданных командой ALTER TABLE ... ADD CONSTRAINT ... NO INHERIT), которые игнорируются; при этом все соответствующие ограничения в таблице-потомке не должны быть ненаследуемыми. В настоящее время ограничения UNIQUE, PRIMARY KEY и FOREIGN KEY не учитываются, но в будущем это может измениться.

NO INHERIT *таблица_родитель*

Эта форма удаляет целевую таблицу из списка потомков указанной родительской таблицы. Результаты запросов к родительской таблице после этого не будут включать записи, взятые из целевой таблицы.

OF *имя_типа*

Эта форма связывает таблицу с составным типом, как если бы она была сформирована командой CREATE TABLE OF. При этом список имён и типов столбцов должен точно соответствовать тому, что образует составной тип; отличие возможно в системном столбце oid. Кроме того, таблица не должна быть потомком какой-либо другой таблицы. Эти ограничения гарантируют, что команда CREATE TABLE OF позволит создать таблицу с таким же определением.

NOT OF

Эта форма разрывает связь типизированной таблицы с её типом.

OWNER

Эта форма меняет владельца таблицы, последовательности, представления, материализованного представления или сторонней таблицы на заданного пользователя.

REPLICA IDENTITY

Эта форма меняет информацию, записываемую в журнал предзаписи для идентификации изменяемых или удаляемых строк. В большинстве случаев старое значение в каждом столбце записывается, только если оно отличается от нового, но значение, хранящееся отдельно, записывается всегда, даже если оно не изменилось. Данный параметр действует только при использовании логической репликации.

DEFAULT

Записываются старые значения столбцов первичного ключа, если он есть. Это режим по умолчанию для несистемных таблиц.

USING INDEX *имя_индекса*

Записываются старые значения столбцов, составляющих заданный индекс, который должен быть уникальным, не частичным, не отложенным и включать только столбцы, помеченные NOT NULL. Если этот индекс удалён, поведение будет таким же, как в режиме NOTHING.

FULL

Записываются старые значения всех столбцов в строке.

NOTHING

Информация о старой строке не записывается. Это режим по умолчанию для системных таблиц.

RENAME

Формы RENAME меняют имя таблицы (или индекса, последовательности, представления, материализованного представления или сторонней таблицы), имя отдельного столбца таблицы или имя ограничения таблицы. На хранимые данные это не влияет.

SET SCHEMA

Эта форма перемещает таблицу в другую схему. Вместе с таблицей перемещаются связанные с ней индексы и ограничения, а также последовательности, принадлежащие столбцам таблицы.

ATTACH PARTITION *имя_секции* FOR VALUES *указание_границ_секции*

Эта форма присоединяет существующую таблицу (которая сама по себе может быть секционированной) в качестве секции к целевой таблице, с применением того же синтаксиса *указание_границ_секции*, что и в CREATE TABLE. Указание границ секции должно соответствовать стратегии секционирования и ключу разбиения целевой таблицы. Присоединяемая таблица должна иметь те же столбцы, что и целевая, и никаких других; более того, должны совпадать и типы столбцов. Кроме того, в ней должны быть те же ограничения NOT NULL и CHECK, что и в целевой таблице. В настоящее время ограничения UNIQUE, PRIMARY KEY и FOREIGN KEY не рассматриваются. Если какое-либо из ограничений CHECK присоединяемой таблицы помечено как NO INHERIT, команда выдаст ошибку; такие ограничения нужно будет пересоздать без предложения NO INHERIT.

Если новая секция является обычной таблицей, чтобы убедиться, что ни одна строка в таблице не нарушает ограничение секции, производится полное сканирование таблицы. Такого сканирования можно избежать, добавив перед выполнением этой команды в таблицу

действующее ограничение CHECK, допускающее только такие строки, которые удовлетворяют задаваемому ограничению секции. Наличие этого ограничения позволит определить, что таблицу не нужно сканировать для проверки ограничения секции. Однако это не будет работать, если какие-либо ключи разбиения являются выражениями и секция не принимает значения NULL. При присоединении секции по списку, не принимающей значения NULL, также добавьте ограничение NOT NULL в столбец ключа разбиения, если это не выражение.

Если новая секция является сторонней таблицей, никакая проверка, удовлетворяют ли все строки сторонней таблицы ограничению секции, не выполняется. (Обсуждение ограничений сторонней таблицы вы можете найти в CREATE FOREIGN TABLE.)

DETACH PARTITION *имя_секции*

Эта форма отсоединяет заданную секцию от целевой таблицы. Отсоединяемая секция продолжит существовать как отдельная таблица, и более не будет иметь никаких связей с таблицей, от которой была отсоединена.

Все виды ALTER TABLE, действующие на одну таблицу, кроме RENAME, SET SCHEMA, ATTACH PARTITION и DETACH PARTITION можно объединить в список множественных изменений и применить вместе. Например, можно добавить несколько столбцов и/или изменить тип столбцов в одной команде. Это особенно полезно для больших таблиц, так как вся таблица обрабатывается за один проход.

Выполнить ALTER TABLE может только владелец соответствующей таблицы. Чтобы сменить схему или табличное пространство таблицы, необходимо также иметь право CREATE в новой схеме или табличном пространстве. Чтобы сделать таблицу потомком другой таблицы, нужно быть владельцем и родительской таблицы. Также, чтобы подсоединить таблицу к другой в качестве секции, необходимо быть владельцем подсоединяемой таблицы. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право CREATE в схеме таблицы. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать таблицу. Однако суперпользователь может сменить владельца таблицы в любом случае.) Чтобы добавить столбец, сменить тип столбца или применить предложение OF, необходимо также иметь право USAGE для соответствующего типа данных.

Параметры

IF EXISTS

Не считать ошибкой, если таблица не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) существующей таблицы, подлежащей изменению. Если перед именем таблицы указано ONLY, изменяется только заданная таблица. Без ONLY изменяется и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно также добавить необязательное указание *, чтобы явно обозначить, что изменению подлежат все дочерние таблицы.

имя_столбца

Имя нового или существующего столбца.

новое_имя_столбца

Новое имя существующего столбца.

новое_имя

Новое имя таблицы.

тип_данных

Тип данных нового столбца или новый тип данных существующего столбца.

ограничение_таблицы

Новое ограничение таблицы.

имя_ограничения

Имя нового или существующего ограничения.

CASCADE

Автоматически удалять объекты, зависящие от удаляемого столбца или ограничения (например, представления, содержащие этот столбец), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

RESTRICT

Отказать в удалении столбца или ограничения, если существуют зависящие от них объекты. Это поведение по умолчанию.

имя_триггера

Имя включаемого или отключаемого триггера.

ALL

Отключить или включить все триггеры, принадлежащие таблице. (Для этого требуются права суперпользователя, если в числе этих триггеров оказываются сгенерированные внутрисистемные триггеры исключений, например те, что реализуют ограничения внешнего ключа или отложенные ограничения уникальности и исключений.)

USER

Отключить или включить все триггеры, принадлежащие таблице, за исключением сгенерированных внутрисистемных триггеров исключений, например, тех, что реализуют ограничения внешнего ключа или отложенные ограничения уникальности и исключений.

имя_индекса

Имя существующего индекса.

параметр_хранения

Имя параметра хранения таблицы

значение

Новое значение параметра хранения таблицы. Это может быть число или строка, в зависимости от параметра.

таблица_родитель

Родительская таблица, с которой будет установлена или разорвана связь данной таблицы.

новый_владелец

Имя пользователя, назначаемого новым владельцем таблицы.

новое_табл_пространство

Имя табличного пространства, в которое будет перемещена таблица.

новая_схема

Имя схемы, в которую будет перемещена таблица.

имя_секции

Имя таблицы, присоединяемой в качестве новой секции, или наоборот, отсоединяемой от данной таблицы.

указание_границ_секции

Указание границ для новой секции. Подробнее синтаксис этого указания рассматривается в описании CREATE TABLE.

Замечания

Ключевое слово COLUMN не несёт смысловой нагрузки и может быть опущено.

Когда столбец добавляется с помощью ADD COLUMN, во всех существующих в таблице строках этот столбец инициализируется значением по умолчанию (или NULL, если предложение DEFAULT для столбца отсутствует). Если предложение DEFAULT отсутствует, это сводится только к изменению метаданных, непосредственного изменения данных таблицы не происходит; добавленные значения NULL выводятся при чтении.

Добавление столбца с предложением DEFAULT или изменение типа существующего столбца влечёт за собой перезапись всей таблицы и её индексов. Но возможно исключение при смене типа существующего столбца: если предложение USING не меняет содержимое столбца и старый тип двоично приводится к новому или является неограниченным доменом поверх нового типа, перезапись таблицы не требуется; хотя все индексы с затронутыми столбцами всё же требуется перестроить. При добавлении или удалении системного столбца oid также необходима перезапись всей таблицы. Перестроение больших таблиц и/или их индексов может быть весьма длительной процедурой, которая при этом временно требует вдвое больше места на диске.

Добавление ограничений CHECK или NOT NULL влечёт за собой необходимость просканировать таблицу, чтобы проверить, что все существующие строки удовлетворяют ограничению, но перезаписывать таблицу при этом не требуется.

Подобным образом, при присоединении новой секции может производиться её сканирование для проверки, соответствуют ли существующие строки ограничению секции.

Возможность объединения множества изменений в одну команду ALTER TABLE полезна в основном тем, что позволяет совместить сканирования и перезаписи таблицы, требуемые этим операциям, и выполнить их за один проход.

Сканирование большой таблицы для проверки нового внешнего ключа или ограничения-проверки может занять длительное время и будет препятствовать внесению других изменений до фиксирования команды ALTER TABLE ADD CONSTRAINT. Основное предназначение указания NOT VALID при добавлении ограничения состоит в уменьшении влияния этой операции на параллельные изменения данных. С указанием NOT VALID команда ADD CONSTRAINT не сканирует таблицу и может быть зафиксирована немедленно. После этого можно выполнить команду VALIDATE CONSTRAINT, которая проверит все существующие строки на соответствие ограничению. Эта команда не будет препятствовать параллельным изменениям, так как ей известно, что в других транзакциях для добавляемых или изменяемых строк ограничение уже будет действовать; проверить нужно только уже существующие строки. Таким образом, для этой проверки в таблице затребуется только блокировка SHARE UPDATE EXCLUSIVE. (Если ограничение является внешним ключом, то в целевой таблице этого ключа также затребуется блокировка ROW SHARE.) Помимо оптимизации параллельной работы, указание NOT VALID и предложение VALIDATE CONSTRAINT полезно в случаях, когда заведомо известно, что в таблице есть строки, нарушающие ограничения. После создания ограничения добавить новые недопустимые строки будет невозможно, а все существующие проблемы могут разрешаться в удобное время, пока VALIDATE CONSTRAINT не выполнится успешно.

Форма DROP COLUMN не удаляет столбец физически, а просто делает его невидимым для операций SQL. При последующих операциях добавления или изменения в этот столбец будет записываться значение NULL. Таким образом, удаление столбца выполняется быстро, но при этом размер таблицы на диске не уменьшается, так как пространство, занимаемое удалённым столбцом, не высвобождается. Это пространство будет освобождено со временем, по мере изменения существующих строк. (При удалении системного столбца oid это поведение не наблюдается, так как немедленно выполняется перезапись таблицы.)

Чтобы принудительно высвободить пространство, занимаемое столбцом, который был удалён, можно выполнить одну из форм ALTER TABLE, производящих перезапись всей таблицы. В результате все строки будут воссозданы так, что в удалённом столбце будет содержаться NULL.

Перезаписывающие формы ALTER TABLE небезопасны с точки зрения MVCC. После перезаписи таблица будет выглядеть пустой для параллельных транзакций, если они работают со снимком, полученным до момента перезаписи. За подробностями обратитесь к Разделу 13.5.

В указании USING предложения SET DATA TYPE на самом деле можно записать выражение со старыми значениями строки; то есть, оно может ссылаться как на преобразуемые столбцы, так и на другие. Это позволяет записывать в SET DATA TYPE очень общие преобразования данных. Ввиду такой гибкости, выражение USING не применяется к значению по умолчанию данного столбца (если таковое есть); результат может быть не константным выражением, что требуется для значения по умолчанию. Это означает, что в случае отсутствия явного приведения или присваивания старого типа новому, SET DATA TYPE может не справиться с преобразованием значения по умолчанию, несмотря на то, что применяется предложение USING. В этих случаях нужно удалить значение по умолчанию с помощью DROP DEFAULT, выполнить ALTER TYPE, а затем с помощью SET DEFAULT задать новое подходящее значение по умолчанию. Подобные соображения применимы и в отношении индексов и ограничений с этим столбцом.

Если у таблицы имеются дочерние таблицы, то добавлять, переименовывать столбцы или менять их тип в родительской таблице, не повторяя ту же операцию в дочерних таблицах, нельзя. Это правило гарантирует, что столбцы в дочерних таблицах всегда соответствуют родительской. Подобным образом, нельзя переименовывать ограничение в родительской таблице, не переименовывая его во всех дочерних таблицах, что тоже гарантирует соответствие всех ограничений. И так как выборка из родительской таблицы влечёт за собой выборку из всех потомков, ограничение родителя не может быть помечено как действующее, если оно также не является действующим в этих потомках. Во всех этих случаях команда ALTER TABLE ONLY не будет выполнена.

Рекурсивная операция DROP COLUMN удалит столбец из дочерней таблицы, только если этот столбец не наследуется от каких-то других родителей и никогда не был определён в дочерней таблице независимо. Нерекурсивная операция DROP COLUMN (т. е., ALTER TABLE ONLY ... DROP COLUMN) никогда не удаляет унаследованные столбцы; вместо этого она помечает их как независимо определённые, а не наследуемые. С секционированной таблицей нерекурсивная команда DROP COLUMN выдаст ошибку, так как все секции таблицы должны содержать те же столбцы, что и главная таблица.

Действия для столбцов идентификации (ADD GENERATED, SET и т. д., DROP IDENTITY), а также действия TRIGGER, CLUSTER, OWNER и TABLESPACE никогда не распространяются рекурсивно на дочерние таблицы; то есть они всегда выполняются так, как будто указано ONLY. Операция добавления ограничения выполняется рекурсивно только для ограничений CHECK, не помеченных как NO INHERIT.

Какие-либо изменения таблиц системного каталога не допускаются.

За более подробным описанием допустимых параметров обратитесь к CREATE TABLE. Дополнительно о наследовании можно узнать в Главе 5.

Примеры

Добавление в таблицу столбца типа varchar:

```
ALTER TABLE distributors ADD COLUMN address varchar(30);
```

Удаление столбца из таблицы:

```
ALTER TABLE distributors DROP COLUMN address RESTRICT;
```

Изменение типов двух существующих столбцов в одной операции:

```
ALTER TABLE distributors
```

```
    ALTER COLUMN address TYPE varchar(80),
    ALTER COLUMN name TYPE varchar(100);
```

Смена типа целочисленного столбца, содержащего время в стиле Unix, на тип `timestamp with time zone` с применением предложения `USING`:

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp SET DATA TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second';
```

То же самое, но в случае, когда у столбца есть значение по умолчанию, не приводимое автоматически к новому типу данных:

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp DROP DEFAULT,
    ALTER COLUMN foo_timestamp TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second',
    ALTER COLUMN foo_timestamp SET DEFAULT now();
```

Переименование существующего столбца:

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

Переименование существующей таблицы:

```
ALTER TABLE distributors RENAME TO suppliers;
```

Переименование существующего ограничения:

```
ALTER TABLE distributors RENAME CONSTRAINT zipchk TO zip_check;
```

Добавление в столбец ограничения NOT NULL:

```
ALTER TABLE distributors ALTER COLUMN street SET NOT NULL;
```

Удаление ограничения NOT NULL из столбца:

```
ALTER TABLE distributors ALTER COLUMN street DROP NOT NULL;
```

Добавление ограничения-проверки в таблицу и все её потомки:

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5);
```

Добавление ограничения-проверки только в таблицу, но не в её потомки:

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5) NO
 INHERIT;
```

(Данное ограничение-проверка не будет наследоваться и будущими потомками тоже.)

Удаление ограничения-проверки из таблицы и из всех её потомков:

```
ALTER TABLE distributors DROP CONSTRAINT zipchk;
```

Удаление ограничения-проверки только из самой таблицы:

```
ALTER TABLE ONLY distributors DROP CONSTRAINT zipchk;
```

(Ограничение-проверка остаётся во всех дочерних таблицах.)

Добавление в таблицу ограничения внешнего ключа:

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES
 addresses (address);
```

Добавление в таблицу ограничения внешнего ключа с наименьшим влиянием на работу других:

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES
 addresses (address) NOT VALID;
ALTER TABLE distributors VALIDATE CONSTRAINT distfk;
```

Добавление в таблицу ограничения уникальности (по нескольким столбцам):

```
ALTER TABLE distributors ADD CONSTRAINT dist_id_zipcode_key UNIQUE (dist_id, zipcode);
```

Добавление в таблицу первичного ключа с автоматическим именем (учтите, что в таблице может быть только один первичный ключ):

```
ALTER TABLE distributors ADD PRIMARY KEY (dist_id);
```

Перемещение таблицы в другое табличное пространство:

```
ALTER TABLE distributors SET TABLESPACE fasttablespace;
```

Перемещение таблицы в другую схему:

```
ALTER TABLE myschema.distributors SET SCHEMA yourschema;
```

Пересоздание ограничения первичного ключа без блокировки изменений в процессе перестроения индекса:

```
CREATE UNIQUE INDEX CONCURRENTLY dist_id_temp_idx ON distributors (dist_id);
ALTER TABLE distributors DROP CONSTRAINT distributors_pkey,
    ADD CONSTRAINT distributors_pkey PRIMARY KEY USING INDEX dist_id_temp_idx;
```

Присоединение секции к таблице, разбиваемой по диапазонам:

```
ALTER TABLE measurement
    ATTACH PARTITION measurement_y2016m07 FOR VALUES FROM ('2016-07-01') TO
 ('2016-08-01');
```

Присоединение секции к таблице, разбиваемой по списку:

```
ALTER TABLE cities
    ATTACH PARTITION cities_ab FOR VALUES IN ('a', 'b');
```

Удаление секции из секционированной таблицы:

```
ALTER TABLE measurement
    DETACH PARTITION measurement_y2015m12;
```

Совместимость

Формы `ADD` (без `USING INDEX`), `DROP [COLUMN]`, `DROP IDENTITY`, `RESTART`, `SET DEFAULT`, `SET DATA TYPE` (без `USING`), `SET GENERATED` и `SET` *параметр_последовательности* соответствуют стандарту SQL. Другие формы являются расширениями стандарта SQL, реализованными в Postgres Pro. Кроме того, расширением является возможность указать в одной команде `ALTER TABLE` несколько операций изменения.

`ALTER TABLE DROP COLUMN` позволяет удалить единственный столбец таблицы и оставить таблицу без столбцов. Это является расширением стандарта SQL, который не допускает существование таблиц с нулём столбцов.

См. также

CREATE TABLE

ALTER TABLESPACE

ALTER TABLESPACE — изменить определение табличного пространства

Синтаксис

```
ALTER TABLESPACE имя RENAME TO новое_имя
ALTER TABLESPACE имя OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }
ALTER TABLESPACE имя SET ( параметр_табличного_пространства = значение [, ... ] )
ALTER TABLESPACE имя RESET ( параметр_табличного_пространства [, ... ] )
```

Описание

`ALTER TABLESPACE` может применяться для изменения определения табличного пространства.

Чтобы изменить определение табличного пространства, нужно быть его владельцем. Чтобы сменить владельца, нужно быть непосредственным или опосредованным членом новой роли-владельца. (Заметьте, что суперпользователи наделяются этими правами автоматически.)

Параметры

имя

Имя существующего табличного пространства.

новое_имя

Новое имя табличного пространства. Новое имя не может начинаться с `pg_`, так как такие имена зарезервированы для системных табличных пространств.

новый_владелец

Новый владелец табличного пространства.

параметр_табличного_пространства

Устанавливаемый или сбрасываемый параметр табличного пространства. В настоящее время поддерживаются только параметры `seq_page_cost`, `random_page_cost` и `effective_io_concurrency`. При установке этих значений для заданного табличного пространства будет переопределена обычная оценка стоимости чтения страниц из таблиц в этом пространстве, настраиваемая одноимённым параметром конфигурации (см. seq_page_cost, random_page_cost, effective_io_concurrency). Это может быть полезно, если одно из табличных пространств размещено на диске, который быстрее или медленнее остальной дисковой системы.

Примеры

Переименование табличного пространства `index_space` в `fast_raid`:

```
ALTER TABLESPACE index_space RENAME TO fast_raid;
```

Смена владельца табличного пространства `index_space`:

```
ALTER TABLESPACE index_space OWNER TO mary;
```

Совместимость

Оператор `ALTER TABLESPACE` отсутствует в стандарте SQL.

См. также

CREATE TABLESPACE, DROP TABLESPACE

ALTER TEXT SEARCH CONFIGURATION

ALTER TEXT SEARCH CONFIGURATION — изменить определение конфигурации текстового поиска

Синтаксис

```
ALTER TEXT SEARCH CONFIGURATION имя
    ADD MAPPING FOR тип_фрагмента [, ... ] WITH имя_словаря [, ... ]
ALTER TEXT SEARCH CONFIGURATION имя
    ALTER MAPPING FOR тип_фрагмента [, ... ] WITH имя_словаря [, ... ]
ALTER TEXT SEARCH CONFIGURATION имя
    ALTER MAPPING REPLACE старый_словарь WITH новый_словарь
ALTER TEXT SEARCH CONFIGURATION имя
    ALTER MAPPING FOR тип_фрагмента [, ... ] REPLACE старый_словарь WITH новый_словарь
ALTER TEXT SEARCH CONFIGURATION имя
    DROP MAPPING [ IF EXISTS ] FOR тип_фрагмента [, ... ]
ALTER TEXT SEARCH CONFIGURATION имя RENAME TO новое_имя
ALTER TEXT SEARCH CONFIGURATION имя OWNER TO { новый_владелец | CURRENT_USER |
 SESSION_USER }
ALTER TEXT SEARCH CONFIGURATION имя SET SCHEMA новая_схема
```

Описание

`ALTER TEXT SEARCH CONFIGURATION` изменяет определение конфигурации текстового поиска. Эта команда позволяет настроить сопоставления типов фрагментов со словарями или сменить владельца или имя конфигурации.

Выполнить `ALTER TEXT SEARCH CONFIGURATION` может только владелец соответствующей конфигурации.

Параметры

имя

Имя (возможно, дополненное схемой) существующей конфигурации текстового поиска.

тип_фрагмента

Имя типа фрагмента, выдаваемое при разборе конфигурации.

имя_словаря

Имя словаря текстового поиска, в котором будет искаться указанный тип фрагмента. Если указаны несколько словарей, они просматриваются в порядке перечисления.

старый_словарь

Имя словаря текстового поиска, которое будет заменено в сопоставлении.

новый_словарь

Имя словаря текстового поиска, которое будет подставлено там, где был *старый_словарь*.

новое_имя

Новое имя конфигурации текстового поиска.

новый_владелец

Новый владелец конфигурации текстового поиска.

новая_схема

Новая схема конфигурации текстового поиска.

Форма `ADD MAPPING FOR` настраивает список словарей, которые будут просматриваться в поиске указанных типов фрагментов; если сопоставление для каких-либо типов уже задано, возникнет ошибка. Форма `ALTER MAPPING FOR` делает то же самое, но она сначала удаляет существующее сопоставление для этих типов фрагментов. Формы `ALTER MAPPING REPLACE` подставляют *новый_словарь* вместо *старый_словарь* везде, где упоминается последний. Это выполняется только для указанных типов фрагментов, когда присутствует `FOR`, либо для всех сопоставлений в конфигурации в противном случае. Форма `DROP MAPPING` удаляет все словари для заданных типов фрагментов, в результате чего фрагменты этих типов будут игнорироваться конфигурацией. Если сопоставлений для заданных типов фрагментов нет, возникает ошибка, если только не добавлено указание `IF EXISTS`.

Примеры

В следующем примере словарь `english` заменяется на `swedish` везде, где использовался `english` в конфигурации `my_config`.

```
ALTER TEXT SEARCH CONFIGURATION my_config
  ALTER MAPPING REPLACE english WITH swedish;
```

Совместимость

Оператор `ALTER TEXT SEARCH CONFIGURATION` отсутствует в стандарте SQL.

См. также

CREATE TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION

ALTER TEXT SEARCH DICTIONARY

ALTER TEXT SEARCH DICTIONARY — изменить определение словаря текстового поиска

Синтаксис

```
ALTER TEXT SEARCH DICTIONARY имя (
    параметр [ = значение ] [, ... ]
)
ALTER TEXT SEARCH DICTIONARY имя RENAME TO новое_имя
ALTER TEXT SEARCH DICTIONARY имя OWNER TO { новый_владелец | CURRENT_USER |
 SESSION_USER }
ALTER TEXT SEARCH DICTIONARY имя SET SCHEMA новая_схема
```

Описание

`ALTER TEXT SEARCH DICTIONARY` изменяет определение словаря текстового поиска. Эта команда позволяет изменить параметры словаря, связанные с шаблонами, или сменить владельца или имя словаря.

Выполнить `ALTER TEXT SEARCH DICTIONARY` может только владелец словаря.

Параметры

имя

Имя (возможно, дополненное схемой) существующего словаря текстового поиска.

параметр

Имя параметра шаблона, устанавливаемого для данного словаря.

значение

Новое значение для параметра настройки шаблонов. Если знак равно и значение опущено, предыдущее значение параметра удаляется из словаря, что позволяет вернутся к значению по умолчанию.

новое_имя

Новое имя словаря текстового поиска.

новый_владелец

Новый владелец словаря текстового поиска.

новая_схема

Новая схема словаря текстового поиска.

Параметры настройки шаблонов могут перечисляться в любом порядке.

Примеры

Команда в следующем примере меняет список стоп-слов словаря на базе Snowball. Другие параметры остаются неизменными.

```
ALTER TEXT SEARCH DICTIONARY my_dict ( StopWords = newrussian );
```

Команда в следующем примере меняет параметр, определяющий язык, на `dutch`, и удаляет параметр, задающий список стоп-слов.

```
ALTER TEXT SEARCH DICTIONARY my_dict ( language = dutch, StopWords );
```

Следующая команда «изменяет» определение словаря, на самом деле не меняя ничего.

```
ALTER TEXT SEARCH DICTIONARY my_dict ( dummy );
```

(Это работает потому, что код удаления параметра не считает ошибкой отсутствие такого параметра.) Этот трюк может быть полезен при изменении файлов конфигурации словаря; `ALTER` принудит все существующие сеансы перечитать файлы конфигурации, что в противном случае они не сделают никогда, если прочитали конфигурацию ранее.

Совместимость

Оператор `ALTER TEXT SEARCH DICTIONARY` отсутствует в стандарте SQL.

См. также

CREATE TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY

ALTER TEXT SEARCH PARSER

ALTER TEXT SEARCH PARSER — изменить определение анализатора текстового поиска

Синтаксис

```
ALTER TEXT SEARCH PARSER имя RENAME TO новое_имя
ALTER TEXT SEARCH PARSER имя SET SCHEMA новая_схема
```

Описание

`ALTER TEXT SEARCH PARSER` изменяет определение анализатора текстового поиска. В настоящее время эта команда позволяет только сменить его имя.

Выполнить `ALTER TEXT SEARCH PARSER` может только суперпользователь.

Параметры

имя

Имя (возможно, дополненное схемой) существующего анализатора текстового поиска.

новое_имя

Новое имя анализатора текстового поиска.

новая_схема

Новая схема анализатора текстового поиска.

Совместимость

Оператор `ALTER TEXT SEARCH PARSER` отсутствует в стандарте SQL.

См. также

CREATE TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER

ALTER TEXT SEARCH TEMPLATE

ALTER TEXT SEARCH TEMPLATE — изменить определение шаблона текстового поиска

Синтаксис

```
ALTER TEXT SEARCH TEMPLATE имя RENAME TO новое_имя
ALTER TEXT SEARCH TEMPLATE имя SET SCHEMA новая_схема
```

Описание

`ALTER TEXT SEARCH TEMPLATE` изменяет определение шаблона текстового поиска. В настоящее время эта команда позволяет только сменить его имя.

Выполнить команду `ALTER TEXT SEARCH TEMPLATE` может только суперпользователь.

Параметры

имя

Имя (возможно, дополненное схемой) существующего шаблона текстового поиска.

новое_имя

Новое имя шаблона текстового поиска.

новая_схема

Новая схема шаблона текстового поиска.

Совместимость

Оператор `ALTER TEXT SEARCH TEMPLATE` отсутствует в стандарте SQL.

См. также

CREATE TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE

ALTER TRIGGER

ALTER TRIGGER — изменить определение триггера

Синтаксис

```
ALTER TRIGGER имя ON имя_таблицы RENAME TO новое_имя
ALTER TRIGGER имя ON имя_таблицы DEPENDS ON EXTENSION имя_расширения
```

Описание

`ALTER TRIGGER` меняет свойства существующего триггера. Предложение `RENAME` переименовывает данный триггер, не затрагивая его определение. Предложение `DEPENDS ON EXTENSION` помечает триггер как зависимый от расширения, так что при удалении расширения будет автоматически удаляться и триггер.

Изменять свойства триггера может только владелец таблицы, с которой работает триггер.

Параметры

имя

Имя существующего триггера, подлежащего изменению.

имя_таблицы

Имя таблицы, с которой работает триггер.

новое_имя

Новое имя триггера.

имя_расширения

Имя расширения, от которого будет зависеть триггер.

Замечания

Возможность временно включать или отключать триггер предоставляется командой ALTER TABLE, а не `ALTER TRIGGER`, так как `ALTER TRIGGER` не позволяет удобным образом выразить указание включить или отключить все триггеры таблицы сразу.

Примеры

Переименование существующего триггера:

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

Обозначение триггера как зависимого от расширения:

```
ALTER TRIGGER emp_stamp ON emp DEPENDS ON EXTENSION emplib;
```

Совместимость

`ALTER TRIGGER` — реализованное в Postgres Pro расширение стандарта SQL.

См. также

ALTER TABLE

ALTER TYPE

ALTER TYPE — изменить определение типа

Синтаксис

```
ALTER TYPE имя действие [, ... ]
ALTER TYPE имя OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }
ALTER TYPE имя RENAME ATTRIBUTE имя_атрибута TO новое_имя_атрибута [ CASCADE |
 RESTRICT ]
ALTER TYPE имя RENAME TO новое_имя
ALTER TYPE имя SET SCHEMA новая_схема
ALTER TYPE имя ADD VALUE [ IF NOT EXISTS ] новое_значение_перечисления [ { BEFORE |
 AFTER } соседнее_значение_перечисления ]
ALTER TYPE имя RENAME VALUE существующее_значение_перечисления
 TO новое_значение_перечисления

Где действие может быть следующим:

    ADD ATTRIBUTE имя_атрибута тип_данных [ COLLATE правило_сортировки ] [ CASCADE |
 RESTRICT ]
    DROP ATTRIBUTE [ IF EXISTS ] имя_атрибута [ CASCADE | RESTRICT ]
    ALTER ATTRIBUTE имя_атрибута [ SET DATA ] TYPE тип_данных
 [ COLLATE правило_сортировки ] [ CASCADE | RESTRICT ]
```

Описание

`ALTER TYPE` изменяет определение существующего типа. Эта команда имеет несколько разновидностей:

`ADD ATTRIBUTE`

Эта форма добавляет в составной тип новый атрибут с тем же синтаксисом, что и CREATE TYPE.

`DROP ATTRIBUTE [ IF EXISTS ]`

Эта форма удаляет атрибут из составного типа. Если указано `IF EXISTS` и атрибут не существует, это не считается ошибкой. В этом случае выдаётся только замечание.

`SET DATA TYPE`

Эта форма меняет тип атрибута составного типа.

`OWNER`

Эта форма меняет владельца типа.

`RENAME`

Эта форма меняет имя типа или имя отдельного атрибута составного типа.

`SET SCHEMA`

Эта форма переносит тип в другую схему.

`ADD VALUE [ IF NOT EXISTS ] [ BEFORE | AFTER ]`

Эта форма добавляет новое значение в тип-перечисление. Порядок нового значения в перечислении можно указать, добавив `BEFORE` (перед) или `AFTER` (после) с одним из существующих значений. Если такое указание отсутствует, новый элемент добавляется в конец списка значений.

С указанием IF NOT EXISTS, если тип уже содержит новое значение, ошибки не произойдёт: будет выдано замечание и ничего больше. Без этого указания, если такое значение уже представлено, возникнет ошибка.

RENAME VALUE

Эта форма переименовывает значение в типе-перечислении. Позиция значения в порядке перечисления при этом не меняется. Если это значение отсутствует или в перечислении уже есть новое имя, выдаётся ошибка.

Операции ADD ATTRIBUTE, DROP ATTRIBUTE и ALTER ATTRIBUTE можно объединить в один список множественных изменений для параллельного выполнения. Например, в одной команде можно добавить сразу несколько атрибутов и/или изменить тип нескольких атрибутов.

Выполнить ALTER TYPE может только владелец соответствующего типа. Чтобы сменить схему типа, необходимо также иметь право CREATE в новой схеме. Чтобы сменить владельца, необходимо быть непосредственным или опосредованным членом новой роли-владельца, а эта роль должна иметь право CREATE в схеме типа. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать тип. Однако суперпользователь может сменить владельца типа в любом случае.) Чтобы добавить атрибут или изменить тип атрибута, также требуется иметь право USAGE для соответствующего типа данных.

Параметры

имя

Имя (возможно, дополненное схемой) существующего типа, подлежащего изменению.

новое_имя

Новое имя типа.

новый_владелец

Имя пользователя, назначаемого новым владельцем типа.

новая_схема

Новая схема типа.

имя_атрибута

Имя атрибута, подлежащего добавлению, изменению или удалению.

новое_имя_атрибута

Новое имя атрибута

тип_данных

Тип данных добавляемого атрибута, либо новый тип данных изменяемого атрибута.

новое_значение_перечисления

Новое значение добавляется в список значений перечисления или для существующего значения задаётся новое имя. Как и все элементы перечисления, оно должно заключаться в кавычки.

соседнее_значение_перечисления

Существующие значение в перечислении, непосредственно перед или после которого по порядку перечисления будет добавлено новое значение. Как и все элементы перечисления, оно должно заключаться в кавычки.

существующее_значение_перечисления

Существующее значение в перечислении, которое будет переименовано. Как и все элементы перечисления, оно должно заключаться в кавычки.

CASCADE

Автоматически распространять действие операции на типизированные таблицы, имеющий данный тип, и их потомки.

RESTRICT

Отказать в выполнении операции, если изменяемый тип является типом типизированной таблицы. Это поведение по умолчанию.

Замечания

`ALTER TYPE ... ADD VALUE` (форму, добавляющую в тип-перечисление новое значение) нельзя выполнять внутри блока транзакции.

Сравнения с добавленными значениями перечисления иногда бывают медленнее сравнений, в которых задействуются только начальные члены типа-перечисления. Обычно это происходит, только если `BEFORE` или `AFTER` устанавливает порядок нового элемента не в конце списка. Однако иногда это наблюдается даже тогда, когда новое значение добавляется в конец списка (это происходит, если счётчик OID «прокручивается» с момента изначального создания типа-перечисления). Это замедление обычно несущественное, но если это важно, вернуть максимальную производительность можно, удалив и создав заново это перечисление, либо выгрузив копию базы данных и загрузив её вновь.

Примеры

Переименование типа данных:

```
ALTER TYPE electronic_mail RENAME TO email;
```

Смена владельца типа `email` на `joe`:

```
ALTER TYPE email OWNER TO joe;
```

Смена схемы типа `email` на `customers`:

```
ALTER TYPE email SET SCHEMA customers;
```

Добавление в тип нового атрибута:

```
ALTER TYPE compfoo ADD ATTRIBUTE f3 int;
```

Добавление нового значения в тип-перечисление, в определённое положение по порядку:

```
ALTER TYPE colors ADD VALUE 'orange' AFTER 'red';
```

Переименование значения в перечислении:

```
ALTER TYPE colors RENAME VALUE 'purple' TO 'mauve';
```

Совместимость

Формы команды, предназначенные для добавления и удаления атрибутов, являются частью стандарта SQL; другие формы относятся к расширениям Postgres Pro.

См. также

CREATE TYPE, DROP TYPE

ALTER USER

ALTER USER — изменить роль в базе данных

Синтаксис

```
ALTER USER указание_роли [ WITH ] параметр [ ... ]

Здесь параметр:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT предел_подключений
    | [ ENCRYPTED ] PASSWORD 'пароль'
    | VALID UNTIL 'дата_время'

ALTER USER имя RENAME TO новое_имя

ALTER USER { указание_роли | ALL } [ IN DATABASE имя_бд ] SET параметр_конфигурации
 { TO | = } { значение | DEFAULT }
ALTER USER { указание_роли | ALL } [ IN DATABASE имя_бд ] SET параметр_конфигурации
 FROM CURRENT
ALTER USER { указание_роли | ALL } [ IN DATABASE имя_бд ] RESET параметр_конфигурации
ALTER USER { указание_роли | ALL } [ IN DATABASE имя_бд ] RESET ALL

Здесь указание_роли:

    имя_роли
  | CURRENT_USER
  | SESSION_USER
```

Описание

Оператор `ALTER USER` теперь стал синонимом оператора ALTER ROLE.

Совместимость

Оператор `ALTER USER` является расширением Postgres Pro. В стандарте SQL определение пользователей считается зависимым от реализации.

См. также

ALTER ROLE

ALTER USER MAPPING

ALTER USER MAPPING — изменить определение сопоставления пользователей

Синтаксис

```
ALTER USER MAPPING FOR { имя_пользователя | USER | CURRENT_USER | SESSION_USER |
 PUBLIC }
    SERVER имя_сервера
    OPTIONS ( [ ADD | SET | DROP ] параметр ['значение'] [, ... ] )
```

Описание

`ALTER USER MAPPING` изменяет определение сопоставления пользователей.

Владелец стороннего сервера может изменить сопоставление любых пользователей на этом сервере. Кроме того, пользователь может изменить сопоставление для своего собственного имени пользователя, если он наделён правом `USAGE` на данном сервере.

Параметры

имя_пользователя

Имя пользователя для сопоставления. Значения `CURRENT_USER` и `USER` соответствуют имени текущего пользователя. Значение `PUBLIC` соответствует именам всех текущих и будущих пользователей системы.

имя_сервера

Имя сервера, для которого меняется сопоставление пользователей.

`OPTIONS ( [ ADD | SET | DROP ] параметр ['значение'] [, ... ] )`

Эти формы меняют параметры сопоставления пользователей. Новые параметры переопределяют любые определённые ранее. Возможные операции с параметрами: `ADD` (добавить), `SET` (установить) и `DROP` (удалить). По умолчанию подразумевается `ADD`. Имена параметров должны быть уникальными; кроме того, они проверяются обёрткой сторонних данных.

Примеры

Изменение пароля в сопоставлении пользователя `bob` на сервере `foo`:

```
ALTER USER MAPPING FOR bob SERVER foo OPTIONS (SET password 'public');
```

Совместимость

`ALTER USER MAPPING` соответствует стандарту ISO/IEC 9075-9 (SQL/MED). Однако есть небольшое синтаксическое различие: в стандарте ключевое слово `FOR` опускается. Но так как и в `CREATE USER MAPPING`, и в `DROP USER MAPPING` слово `FOR` находится в аналогичных позициях, а IBM DB2 (ещё одна популярная реализация SQL/MED) требует его и для `ALTER USER MAPPING`, Postgres Pro в этом аспекте отклоняется от стандарта ради согласованности и совместимости.

См. также

CREATE USER MAPPING, DROP USER MAPPING

ALTER VIEW

ALTER VIEW — изменить определение представления

Синтаксис

```
ALTER VIEW [ IF EXISTS ] имя ALTER [ COLUMN ] имя_столбца SET DEFAULT выражение
ALTER VIEW [ IF EXISTS ] имя ALTER [ COLUMN ] имя_столбца DROP DEFAULT
ALTER VIEW [ IF EXISTS ] имя OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }
ALTER VIEW [ IF EXISTS ] имя RENAME TO новое_имя
ALTER VIEW [ IF EXISTS ] имя SET SCHEMA новая_схема
ALTER VIEW [ IF EXISTS ] имя SET ( имя_параметра_представления
 [= значение_параметра_представления] [, ... ] )
ALTER VIEW [ IF EXISTS ] имя RESET ( имя_параметра_представления [, ... ] )
```

Описание

`ALTER VIEW` изменяет различные дополнительные свойства представления. (Для изменения запроса, определяющего представление, используйте команду `CREATE OR REPLACE VIEW`.)

Выполнить `ALTER VIEW` может только владелец представления. Чтобы сменить схему представления, необходимо также иметь право `CREATE` в новой схеме. Чтобы сменить владельца, требуется также быть непосредственным или опосредованным членом новой роли, а эта роль должна иметь право `CREATE` в схеме представления. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать представление. Однако суперпользователь может сменить владельца представления в любом случае.)

Параметры

имя

Имя существующего представления (возможно, дополненное схемой).

`IF EXISTS`

Не считать ошибкой, если представление не существует. В этом случае будет выдано замечание.

`SET`/`DROP DEFAULT`

Эти формы устанавливают или удаляют значение по умолчанию в заданном столбце. Значение по умолчанию подставляется в команды `INSERT` и `UPDATE`, вносящие данные в представление, до применения каких-либо правил или триггеров в этом представлении. Таким образом, значения по умолчанию в представлении имеют приоритет перед значениями по умолчанию в нижележащих отношениях.

новый_владелец

Имя пользователя, назначаемого новым владельцем представления.

новое_имя

Новое имя представления.

новая_схема

Новая схема представления.

`SET ( `*имя_параметра_представления* `[= `*значение_параметра_представления*`] [, ... ] )`
`RESET ( `*имя_параметра_представления* `[, ... ] )`

Устанавливает или сбрасывает параметры представления. В настоящее время поддерживаются параметры:

`check_option` (`string`)

Изменяет параметр проверки представления. Допустимые значения: `local` (локальная) или `cascaded` (каскадная).

`security_barrier` (`boolean`)

Изменяет свойство представления, включающее барьер безопасности. Значение должно быть логическим: `true` или `false`.

Замечания

По историческим причинам команду `ALTER TABLE` можно использовать и с представлениями; но единственно допустимые для работы с представлениями вариации `ALTER TABLE` равносильны вышеперечисленным командам.

Примеры

Переименование представления `foo` в `bar`:

```
ALTER VIEW foo RENAME TO bar;
```

Добавление значения столбца по умолчанию в изменяемое представление:

```
CREATE TABLE base_table (id int, ts timestamptz);
CREATE VIEW a_view AS SELECT * FROM base_table;
ALTER VIEW a_view ALTER COLUMN ts SET DEFAULT now();
INSERT INTO base_table(id) VALUES(1);  -- в ts окажется значение NULL
INSERT INTO a_view(id) VALUES(2);  -- в ts окажется текущее время
```

Совместимость

`ALTER VIEW` — реализованное в Postgres Pro расширение стандарта SQL.

См. также

CREATE VIEW, DROP VIEW

ANALYZE

ANALYZE — собрать статистику по базе данных

Синтаксис

```
ANALYZE [ VERBOSE ] [ имя_таблицы [ ( имя_столбца [, ...] ) ] ]
```

Описание

`ANALYZE` собирает статистическую информацию о содержимом таблиц в базе данных и сохраняет результаты в системном каталоге `pg_statistic`. Впоследствии планировщик запросов будет использовать эту статистику для выбора наиболее эффективных планов выполнения запросов.

Без параметров команда `ANALYZE` анализирует все таблицы в текущей базе данных. Если в параметрах передано имя таблицы, `ANALYZE` обрабатывает только заданную таблицу. Также эта команда принимает список имён столбцов, что позволяет запустить сбор статистики только по этим столбцам.

Параметры

`VERBOSE`

Включает вывод сообщений о процессе выполнения.

имя_таблицы

Имя (возможно, дополненное схемой) определённой таблицы, подлежащей анализу. Если опущено, анализироваться будут все обычные и секционированные таблицы, а также материализованные представления в текущей базе данных (но не сторонние таблицы). Если задано имя секционированной таблицы, обновлена будет как статистика наследования для этой таблицы, так и статистика отдельных её секций.

имя_столбца

Имя столбца, подлежащего анализу. По умолчанию анализируются все столбцы.

Выводимая информация

С указанием `VERBOSE` команда `ANALYZE` выдаёт сообщения о процессе анализа, отмечая текущую обрабатываемую таблицу. Также она выводит различные статистические сведения о таблицах.

Замечания

Чтобы осуществить анализ таблицы, обычно нужно быть владельцем этой таблицы или суперпользователем. Однако владельцам баз данных также разрешено выполнять анализ всех таблиц в своих базах, за исключением общих каталогов. (Ограничение в отношении общих каталогов означает, что действительно глобальную команду `ANALYZE` может выполнить только суперпользователь.) `ANALYZE` при обработке пропускает все таблицы, на очистку которых текущий пользователь не имеет прав.

Сторонние таблицы анализируются только при явном указании и только если соответствующая обёртка сторонних данных поддерживает команду `ANALYZE`. Если эта команда не поддерживается, при выполнении `ANALYZE` выводится предупреждение и больше ничего не происходит.

В стандартной конфигурации Postgres Pro работающий демон автоочистки (см. Подраздел 23.1.6) запускает анализ таблиц автоматически, когда они изначально заполняются данными, и периодически, по мере того, как они меняются. Если автоочистка отключена, рекомендуется запускать `ANALYZE` время от времени, либо после кардинальных изменений в таблице. Точная

статистика помогает планировщику выбрать наиболее эффективный план запроса и тем самым увеличивает скорость выполнения запроса. Обычно для баз, где данные в основном читаются, выполняют VACUUM и ANALYZE раз в день, во время наименьшей активности. (Этого будет недостаточно, если данные меняются очень активно.)

ANALYZE запрашивает для целевой таблицы блокировку только на чтение, так что эта команда может выполняться параллельно с другими операциями с таблицей.

Статистика, собираемая командой ANALYZE, обычно включает список из нескольких самых частых значений в каждом столбце и гистограмму, отражающую примерное распределение данных во всех столбцах. Один или оба этих элемента статистики могут быть опущены, если ANALYZE сочтёт их неинтересными (например, в столбце уникального ключа нет повторяющихся значений), либо если тип данных столбца не поддерживает соответствующие операторы. Более подробно статистика описывается в Главе 23.

В больших таблицах ANALYZE не просматривает все строки, а обрабатывает только небольшую случайную выборку. Это позволяет проанализировать за короткое время даже очень большие таблицы. Однако учтите, что такая статистика будет лишь приблизительной и может немного меняться при каждом выполнении ANALYZE, даже если фактическое содержимое таблицы остаётся неизменным. Это может приводить к небольшим изменениям в оценках стоимости запросов, выводимых командой EXPLAIN. В редких случаях вследствие этой недетерминированности планировщик меняет свой выбор после выполнения ANALYZE. Чтобы избежать этого, увеличьте объём статистики, собираемой командой ANALYZE, как описано ниже.

Количеством статистики можно управлять, настраивая конфигурационную переменную default_statistics_target или устанавливая ориентир статистики на уровне столбцов командой ALTER TABLE ... ALTER COLUMN ... SET STATISTICS (см. ALTER TABLE). Ориентир задаёт максимальное число записей в списке наиболее распространённых значений и максимальное число интервалов в гистограмме. По умолчанию значение ориентира равно 100, но его можно увеличить или уменьшить в поисках баланса между точностью оценок планировщика и временем, требующимся для выполнения ANALYZE, а также объёмом статистики в таблице pg_statistic. Если установить ориентир статистики равным нулю, статистика по таким столбцам собираться не будет. Это может быть полезно для столбцов, которые никогда не фигурируют в предложениях WHERE, GROUP BY и ORDER BY, так как планировщик никогда не будет использовать их статистику.

Число строк таблицы, выбираемых для подготовки статистики, определяется наибольшим ориентиром статистики по всем анализируемым столбцам этой таблицы. Увеличение ориентира приводит к пропорциональному увеличению времени и пространства, требуемого для выполнения ANALYZE.

Одним из показателей, оцениваемых командой ANALYZE, является число различных значений, встречающихся в каждом столбце. Так как рассматривается только подмножество всех строк, эта оценка иногда может быть весьма неточной, даже при самых больших ориентирах статистики. Если эта неточность приводит к плохому выбору плана запроса, более точное значение можно определить вручную и затем задать его командой ALTER TABLE ... ALTER COLUMN ... SET (n_distinct = ...) (см. ALTER TABLE).

Если у анализируемой таблицы есть потомки, ANALYZE соберёт два набора статистик: один по строкам только родительской таблицы, а второй по строкам родительской и всех дочерних таблиц. Второй набор статистики необходим для планирования запросов, обращающихся к дереву наследования как одному целому. Сами дочерние таблицы в этом случае отдельно не анализируются. Демон автоочистки, однако, принимая решение об автоматическом запуске анализа, будет учитывать операции добавления или изменения данных только в самой родительской таблице. Если именно в этой таблице изменение и добавление происходят редко, наследуемая статистика может терять актуальность, если не запускать ANALYZE вручную.

Для секционированных таблиц ANALYZE собирает статистику, обрабатывая выборки строки из всех секций, а также рекурсивно обновляет статистику во всех секциях. Даже в многоуровневой иерархии секционирования каждая конечная секция анализируется только один раз. Отдельно

для родительских таблиц (без рассмотрения данных их секций) статистика не собирается, так как секционирование гарантирует, что они пустые.

Процесс автоочистки не затрагивает секционированные таблицы и родительские таблицы в иерархии наследования, если изменения происходят только в дочерних таблицах. Поэтому, чтобы статистика всей иерархии наследования была актуальной, обычно необходимо периодически выполнять `ANALYZE` вручную.

Если какие-либо из дочерних таблиц или секций являются сторонними таблицами и их обёртки сторонних данных не поддерживают `ANALYZE`, эти дочерние таблицы игнорируются при сборе статистики наследования.

Если анализируемая таблица оказалась пустой, `ANALYZE` не будет обновлять статистику по этой таблице; в базе сохранится статистика, собранная ранее.

Совместимость

Оператор `ANALYZE` отсутствует в стандарте SQL.

См. также

VACUUM, vacuumdb, Подраздел 18.4.4, Подраздел 23.1.6

BEGIN

BEGIN — начать блок транзакции

Синтаксис

```
BEGIN [ WORK | TRANSACTION ] [ режим_транзакции [, ...] ]

Где режим_транзакции может быть следующим:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
 UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

Описание

`BEGIN` начинает блок транзакции, то есть обозначает, что все операторы после команды `BEGIN` и до явной команды COMMIT или ROLLBACK будут выполняться в одной транзакции. По умолчанию (без `BEGIN`) Postgres Pro выполняет транзакции в режиме «autocommit» (автофиксация), то есть каждый оператор выполняется в своей отдельной транзакции, которая неявно фиксируется в конце оператора (если оператор был выполнен успешно; в противном случае транзакция откатывается).

В блоке транзакции операторы выполняются быстрее, так как для запуска/фиксации транзакции производится масса операций, нагружающих процессор и диск. Кроме того, выполнение нескольких операторов в одной транзакции позволяет обеспечить целостность при внесении серии связанных изменений; другие сеансы не видят промежуточное состояние, когда произошли ещё не все связанные изменения.

Если указан уровень изоляции, режим чтения/записи или устанавливается отложенный режим, новая транзакция получает те же характеристики, что и после выполнения SET TRANSACTION.

Параметры

`WORK`
`TRANSACTION`

Необязательные ключевые слова, не оказывают никакого влияния.

За описанием других параметров обратитесь к SET TRANSACTION.

Замечания

START TRANSACTION делает то же, что и `BEGIN`.

Для завершения блока транзакции используйте COMMIT или ROLLBACK.

При попытке выполнить `BEGIN` внутри уже начатого блока транзакции будет выдано предупреждение, а состояние транзакции не изменится. Для вложения подтранзакций внутри блока транзакций используйте точки сохранения (см. SAVEPOINT).

Для сохранения обратной совместимости допускается перечисление *режимов_транзакции* без запятых.

Примеры

Начало блока транзакции:

```
BEGIN;
```

Совместимость

`BEGIN` — это языковое расширение Postgres Pro. Эта команда равнозначна соответствующей стандарту SQL команде START TRANSACTION, на справочной странице которой можно найти дополнительные сведения о совместимости.

Значение `DEFERRABLE` параметра *`режим_транзакции`* является языковым расширением Postgres Pro.

По стечению обстоятельств ключевое слово `BEGIN` имеет другое значение во встраиваемом SQL, поэтому при портировании приложений баз данных рекомендуется внимательно сверить семантику транзакций.

См. также

COMMIT, ROLLBACK, START TRANSACTION, SAVEPOINT

CHECKPOINT

CHECKPOINT — произвести контрольную точку в журнале предзаписи

Синтаксис

```
CHECKPOINT
```

Описание

Контрольная точка — это момент в последовательности событий в журнале предзаписи, когда все файлы данных приводятся в актуальное состояние, соответствующее информации в журнале. При этом все файлы данных сохраняются на диск. Более подробно о том, что происходит во время контрольной точки, можно узнать в Разделе 29.4.

Команда `CHECKPOINT` приводит к принудительному выполнению контрольной точки в момент вызова, не дожидаясь периодической контрольной точки, производимой системой по графику (это настраивается параметрами в Подраздел 18.5.2). Команда `CHECKPOINT` не предназначена для применения при обычном использовании базы данных.

Если `CHECKPOINT` выполняется в процессе восстановления, вместо записи новой контрольной точки производится точка перезапуска (см. Раздел 29.4).

Выполнять `CHECKPOINT` могут только суперпользователи.

Совместимость

Команда `CHECKPOINT` является языковым расширением Postgres Pro.

CLOSE

CLOSE — закрыть курсор

Синтаксис

```
CLOSE { имя | ALL }
```

Описание

`CLOSE` освобождает ресурсы, связанные с открытым курсором. Когда курсор закрыт, никакие операции с ним невозможны. Закрывать курсор следует, когда он становится ненужным.

Все не удерживаемые открытые курсоры закрываются неявно при завершении транзакции командами `COMMIT` или `ROLLBACK`. Удерживаемый курсор закрывается неявно, если транзакция, его создавшая, прерывается командой `ROLLBACK`. Если создавшая его транзакция завершается успешной фиксацией, удерживаемый курсор остаётся открытым до явного вызова команды `CLOSE` или отключения клиента.

Параметры

имя

Имя открытого курсора, который будет закрыт.

`ALL`

Закрывает все открытые курсоры.

Замечания

В Postgres Pro нет явной команды `OPEN` для курсора; курсор считается открытым при объявлении. Чтобы объявить курсор, используйте оператор DECLARE.

Получить список всех доступных курсоров можно, обратившись к системному представлению `pg_cursors`.

Если курсор был закрыт после точки сохранения, а затем произошёл откат к этой точке, действие команды `CLOSE` не отменяется; то есть курсор остаётся закрытым.

Примеры

Следующая команда закрывает курсор `liahona`:

```
CLOSE liahona;
```

Совместимость

Оператор `CLOSE` полностью соответствует стандарту SQL. `CLOSE ALL` является расширением Postgres Pro.

См. также

DECLARE, FETCH, MOVE

CLUSTER

CLUSTER — кластеризовать таблицу согласно индексу

Синтаксис

```
CLUSTER [VERBOSE] имя_таблицы [ USING имя_индекса ]
CLUSTER [VERBOSE]
```

Описание

Оператор `CLUSTER` указывает Postgres Pro кластеризовать таблицу, заданную параметром *имя_таблицы*, согласно индексу, заданному параметром *имя_индекса*. Указанный индекс уже должен быть определён в таблице *имя_таблицы*.

В результате кластеризации таблицы её содержимое физически переупорядочивается в зависимости от индекса. Кластеризация является одноразовой операцией: последующие изменения в таблице нарушают порядок кластеризации. Другими словами, система не пытается автоматически сохранять порядок новых или изменённых строк в соответствии с индексом. (Если такое желание возникает, можно периодически повторять кластеризацию, выполняя команду снова. Кроме того, если для заданной таблицы установить параметр `FILLFACTOR` меньше 100%, это может помочь сохранить порядок кластеризации при изменениях, так как изменяемые строки будут помещаться в ту же страницу, если в ней достаточно места.)

Когда таблица кластеризована, Postgres Pro запоминает, по какому именно индексу. Форма `CLUSTER` *имя_таблицы* повторно кластеризует таблицу по тому же индексу. Для установки индекса, который будет использоваться для будущих операций кластеризации, или очистки предыдущего значения можно также применить команду `CLUSTER` или формы `SET WITHOUT CLUSTER` команды ALTER TABLE.

`CLUSTER` без параметров повторно кластеризует все ранее кластеризованные таблицы в текущей базе данных, принадлежащие пользователю, вызывающему команду, или все такие таблицы, если её вызывает суперпользователь. Эту форму `CLUSTER` нельзя выполнять внутри блока транзакции.

В процессе кластеризации таблицы для неё запрашивается блокировка `ACCESS EXCLUSIVE`. Это препятствует выполнению всех других операций (чтению и записи) с таблицей до завершения `CLUSTER`.

Параметры

имя_таблицы

Имя таблицы (возможно, дополненное схемой).

имя_индекса

Имя индекса.

`VERBOSE`

Выводит отчёт о процессе кластеризации по мере обработки таблиц.

Замечания

В случаях, когда происходит обращение к случайным единичным строкам таблицы, фактический порядок данных в этой таблице не важен. Но если обращения к одним данным происходят чаще, чем к другим, и есть индекс, который собирает их вместе, применение команды `CLUSTER` может быть полезным. Например, когда из таблицы запрашивается диапазон индексированных значений, либо одно индексированное значение, которому соответствуют несколько строк, `CLUSTER` может

помочь, так как страница таблицы, найденная по индексу для первой искомой строки, скорее всего, будет содержать и все остальные искомые строки. Таким образом, кластеризация помогает соптимизировать обращения к диску и ускорить запросы.

`CLUSTER` может переупорядочить таблицу, выполнив либо сканирование указанного индекса, либо (для индексов-B-деревьев) последовательное сканирование, а затем сортировку. Наилучший по скорости вариант будет выбран, исходя из имеющейся статистической информации и параметров планировщика.

Когда выбирается сканирование индекса, создаётся временная таблица, содержащая данные целевой таблицы по порядку индекса. Также создаются копии всех индексов таблицы. Таким образом, для этой операции требуется объём дискового пространства не меньше, чем размер таблицы и индексов в сумме.

В случае выбора последовательного сканирования и сортировки создаётся ещё и временный файл для сортировки, так что пиковым требованием будет удвоенный размер таблицы плюс размер индексов. Этот метод часто быстрее, чем сканирование по индексу, но если требование к дисковому пространству неприемлемо, можно отключить его выбор, временно установив enable_sort в `off`.

Перед кластеризацией рекомендуется установить в maintenance_work_mem достаточно большое значение (но не больше, чем объём ОЗУ, который вы хотите выделить для операции `CLUSTER`).

Так как планировщик записывает статистику, связанную с порядком таблиц, для вновь кластеризуемых таблиц рекомендуется запускать ANALYZE. В противном случае планировщик может ошибиться с выбором плана запроса.

Так как `CLUSTER` запоминает, по каким индексам кластеризованы таблицы, достаточно лишь один раз вручную кластеризовать нужные таблицы, а затем настроить периодический скрипт обслуживания, который будет выполнять `CLUSTER` без параметров, с тем чтобы эти таблицы регулярно кластеризовались.

Примеры

Кластеризация таблицы `employees` согласно её индексу `employees_ind`:

```
CLUSTER employees USING employees_ind;
```

Кластеризация таблицы `employees` согласно тому же индексу, что был использован ранее:

```
CLUSTER employees;
```

Кластеризация всех таблиц в базе данных, что были кластеризованы ранее:

```
CLUSTER;
```

Совместимость

Оператор `CLUSTER` отсутствует в стандарте SQL.

Синтаксис

`CLUSTER` *`имя_индекса`* `ON` *`имя_таблицы`*

так же является допустимым для совместимости с PostgreSQL до версии 8.3.

См. также

clusterdb

COMMENT

COMMENT — задать или изменить комментарий объекта

Синтаксис

```
COMMENT ON
{
  ACCESS METHOD имя_объекта |
  AGGREGATE имя_агрегатной_функции ( сигнатура_агр_функции ) |
  CAST (исходный_тип AS целевой_тип) |
  COLLATION имя_объекта |
  COLUMN имя_отношения.имя_столбца |
  CONSTRAINT имя_ограничения ON имя_таблицы |
  CONSTRAINT имя_ограничения ON DOMAIN имя_домена |
  CONVERSION имя_объекта |
  DATABASE имя_объекта |
  DOMAIN имя_объекта |
  EXTENSION имя_объекта |
  EVENT TRIGGER имя_объекта |
  FOREIGN DATA WRAPPER имя_объекта |
  FOREIGN TABLE имя_объекта |
  FUNCTION имя_функции [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ] |
  INDEX имя_объекта |
  LARGE OBJECT oid_большого_объекта |
  MATERIALIZED VIEW имя_объекта |
  OPERATOR имя_оператора (тип_слева, тип_справа) |
  OPERATOR CLASS имя_объекта USING индексный_метод |
  OPERATOR FAMILY имя_объекта USING индексный_метод |
  POLICY имя_политики ON имя_таблицы |
  [ PROCEDURAL ] LANGUAGE имя_объекта |
  PUBLICATION имя_объекта |
  ROLE имя_объекта |
  RULE имя_правила ON имя_таблицы |
  SCHEMA имя_объекта |
  SEQUENCE имя_объекта |
  SERVER имя_объекта |
  STATISTICS имя_объекта |
  SUBSCRIPTION имя_объекта |
  TABLE имя_объекта |
  TABLESPACE имя_объекта |
  TEXT SEARCH CONFIGURATION имя_объекта |
  TEXT SEARCH DICTIONARY имя_объекта |
  TEXT SEARCH PARSER имя_объекта |
  TEXT SEARCH TEMPLATE имя_объекта |
  TRANSFORM FOR имя_типа LANGUAGE имя_языка |
  TRIGGER имя_триггера ON имя_таблицы |
  TYPE имя_объекта |
  VIEW имя_объекта
} IS 'текст'

Здесь сигнатура_агр_функции:

* |
[ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] |
```

```
[ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] ] ORDER BY
  [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ]
```

Описание

`COMMENT` сохраняет комментарий об объекте базы данных.

Для каждого объекта сохраняется только одна строка, так что для изменения комментария нужно просто выполнить `COMMENT` ещё раз для того же объекта. Чтобы удалить комментарий, вместо текстовой строки укажите `NULL`. При удалении объектов комментарии удаляются автоматически.

Для целевого объекта запрашивается блокировка `SHARE UPDATE EXCLUSIVE`.

Для большинства типов объектов комментарий может установить только владелец объекта. Но так как роли не имеют владельцев, `COMMENT ON ROLE` для ролей суперпользователей разрешено выполнять только суперпользователям, а для обычных ролей — тем, кто имеет право `CREATEROLE`. Так же не имеют владельцев и методы доступа; чтобы добавить комментарий для метода доступа, нужно быть суперпользователем. Разумеется, суперпользователи могут задавать комментарии для любых объектов.

Просмотреть комментарии можно в psql, используя семейство команд `\d`. Имеется возможность получать комментарии и в других пользовательских интерфейсах, используя те же встроенные функции, что использует psql, а именно `obj_description`, `col_description` и `shobj_description` (см. Таблицу 9.68).

Параметры

имя_объекта
имя_отношения.имя_столбца
имя_агрегатной_функции
имя_ограничения
имя_функции
имя_оператора
имя_политики
имя_правила
имя_триггера

Имя объекта, для которого задаётся комментарий. Имена таблиц, агрегатных функций, правил сортировки, перекодировок, доменов, сторонних таблиц, функций, индексов, операторов, классов и семейств операторов, последовательностей, объектов статистики, объектов текстового поиска, типов и представлений могут быть дополнены именем схемы. При определении комментария для столбца, *имя_отношения* должно ссылаться на таблицу, представление, составной тип или стороннюю таблицу.

имя_таблицы
имя_домена

При создании комментария для ограничения, триггера, правила или политики эти параметры задают имя таблицы или домена, к которым относится этот объект.

исходный_тип

Имя исходного типа данных для приведения.

целевой_тип

Имя целевого типа данных для приведения.

режим_аргумента

Режим аргумента обычной или агрегатной функции: `IN`, `OUT`, `INOUT` или `VARIADIC`. По умолчанию подразумевается `IN`. Заметьте, что `COMMENT` не учитывает аргументы `OUT`, так как

для идентификации функции нужны только типы входных аргументов. Поэтому достаточно перечислить только аргументы IN, INOUT и VARIADIC.

имя_аргумента

Имя аргумента обычной или агрегатной функции. Заметьте, что на самом деле COMMENT не обращает внимание на имена аргументов, так как для однозначной идентификации функции достаточно только типов аргументов.

тип_аргумента

Тип данных аргумента обычной или агрегатной функции.

oid_большого_объекта

OID большого объекта.

тип_слева
тип_справа

Тип данных аргументов оператора (возможно, дополненный именем схемы). В случае отсутствия аргумента префиксного или постфиксного оператора укажите вместо типа NONE.

PROCEDURAL

Это слово не несёт смысловой нагрузки.

имя_типа

Имя типа данных, для которого предназначена трансформация.

имя_языка

Имя языка, для которого предназначена трансформация.

текст

Новый комментарий, записанный в виде строковой константы (или NULL для удаления комментария).

Замечания

В настоящее время механизм безопасности в части просмотра комментариев отсутствует: любой пользователь, подключённый к базе данных, может видеть все комментарии всех объектов базы. Для общих объектов, таких как базы данных, роли и табличные пространства, комментарии хранятся глобально, так что их может видеть любой пользователь, подключённый к любой базе данных в кластере. Поэтому ничего секретного писать в комментариях не следует.

Примеры

Добавление комментария для таблицы mytable:

```
COMMENT ON TABLE mytable IS 'Это моя таблица.';
```

Удаление его:

```
COMMENT ON TABLE mytable IS NULL;
```

Ещё несколько примеров:

```
COMMENT ON ACCESS METHOD rtree IS 'Метод доступа R-Tree';
COMMENT ON AGGREGATE my_aggregate (double precision) IS 'Вычисляет дисперсию выборки';
COMMENT ON CAST (text AS int4) IS 'Выполняет приведение строк к int4';
COMMENT ON COLLATION "fr_CA" IS 'Канадский французский';
COMMENT ON COLUMN my_table.my_column IS 'Порядковый номер сотрудника';
COMMENT ON CONVERSION my_conv IS 'Перекодировка в UTF8';
```

```
COMMENT ON CONSTRAINT bar_col_cons ON bar IS 'Ограничение столбца col';
COMMENT ON CONSTRAINT dom_col_constr ON DOMAIN dom IS 'Ограничение col для домена';
COMMENT ON DATABASE my_database IS 'База данных разработчиков';
COMMENT ON DOMAIN my_domain IS 'Домен почтового адреса';
COMMENT ON EXTENSION hstore IS 'Реализует тип данных hstore';
COMMENT ON FOREIGN DATA WRAPPER mywrapper IS 'Моя обёртка сторонних данных';
COMMENT ON FOREIGN TABLE my_foreign_table IS 'Информация о сотрудниках в другой БД';
COMMENT ON FUNCTION my_function (timestamp) IS 'Возвращает число римскими цифрами';
COMMENT ON INDEX my_index IS 'Обеспечивает уникальность по коду сотрудника';
COMMENT ON LANGUAGE plpython IS 'Поддержка Python для хранимых процедур';
COMMENT ON LARGE OBJECT 346344 IS 'Документ планирования';
COMMENT ON MATERIALIZED VIEW my_matview IS 'Сводка истории заказов';
COMMENT ON OPERATOR ^ (text, text) IS 'Вычисляет пересечение двух текстов';
COMMENT ON OPERATOR - (NONE, integer) IS 'Унарный минус';
COMMENT ON OPERATOR CLASS int4ops USING btree IS 'Операторы для четырёхбайтовых целых
 (для B-деревьев)';
COMMENT ON OPERATOR FAMILY integer_ops USING btree IS 'Все целочисленные операторы (для
 B-деревьев)';
COMMENT ON POLICY my_policy ON mytable IS 'Фильтр строк по пользователям';
COMMENT ON ROLE my_role IS 'Административная группа для таблиц бухгалтерии';
COMMENT ON RULE my_rule ON my_table IS 'Протоколирует изменения в записях сотрудников';
COMMENT ON SCHEMA my_schema IS 'Данные отдела';
COMMENT ON SEQUENCE my_sequence IS 'Предназначена для генерации первичных ключей';
COMMENT ON SERVER myserver IS 'Мой сторонний сервер';
COMMENT ON STATISTICS my_statistics IS 'Улучшает оценку числа строк для планировщика';
COMMENT ON TABLE my_schema.my_table IS 'Данные сотрудников';
COMMENT ON TABLESPACE my_tablespace IS 'Табличное пространство для индексов';
COMMENT ON TEXT SEARCH CONFIGURATION my_config IS 'Фильтрация специальных слов';
COMMENT ON TEXT SEARCH DICTIONARY swedish IS 'Стеммер Snowball для шведского языка';
COMMENT ON TEXT SEARCH PARSER my_parser IS 'Разделяет текст на слова';
COMMENT ON TEXT SEARCH TEMPLATE snowball IS 'Стеммер Snowball';
COMMENT ON TRANSFORM FOR hstore LANGUAGE plpythonu IS 'Трансформирует данные из hstore
 в словарь языка Python';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Обеспечивает ссылочную целостность';
COMMENT ON TYPE complex IS 'Тип данных комплексных чисел';
COMMENT ON VIEW my_view IS 'Представление расходов по отделам';
```

Совместимость

Оператор `COMMENT` отсутствует в стандарте SQL.

COMMIT

COMMIT — зафиксировать текущую транзакцию

Синтаксис

```
COMMIT [ WORK | TRANSACTION ]
```

Описание

`COMMIT` фиксирует текущую транзакцию. Все изменения, произведённые транзакцией, становятся видимыми для других и гарантированно сохранятся в случае сбоя.

Параметры

`WORK`
`TRANSACTION`

Необязательные ключевые слова, не оказывают никакого влияния.

Замечания

Для прерывания транзакции используйте ROLLBACK.

При попытке выполнить `COMMIT` вне транзакции ничего не произойдёт, но будет выдано предупреждение.

Примеры

Следующая команда фиксирует текущую транзакцию и сохраняет все изменения:

```
COMMIT;
```

Совместимость

В стандарте SQL описаны только две формы: `COMMIT` и `COMMIT WORK`. В остальном эта команда полностью соответствует стандарту.

См. также

BEGIN, ROLLBACK

COMMIT PREPARED

COMMIT PREPARED — зафиксировать транзакцию, которая ранее была подготовлена для двухфазной фиксации

Синтаксис

```
COMMIT PREPARED id_транзакции
```

Описание

`COMMIT PREPARED` фиксирует транзакцию, находящуюся в подготовленном состоянии.

Параметры

id_транзакции

Идентификатор транзакции, которая будет зафиксирована.

Замечания

Зафиксировать подготовленную транзакцию может либо пользователь, выполнявший её изначально, либо суперпользователь. При этом не обязательно работать в том же сеансе, где выполнялась транзакция.

Эту команду нельзя выполнить внутри блока транзакции. Подготовленная транзакция фиксируется немедленно.

Все существующие в текущий момент подготовленные транзакции показываются в системном представлении `pg_prepared_xacts`.

Примеры

Фиксация транзакции, имеющей идентификатор `foobar`:

```
COMMIT PREPARED 'foobar';
```

Совместимость

Оператор `COMMIT PREPARED` является расширением Postgres Pro. Он предназначен для использования внешними системами управления транзакциями, некоторые из которых работают по стандартам (например, X/Open XA), но сторона SQL в этих системах не стандартизирована.

См. также

PREPARE TRANSACTION, ROLLBACK PREPARED

COPY

COPY — копировать данные между файлом и таблицей

Синтаксис

```
COPY имя_таблицы [ ( имя_столбца [, ...] ) ]
    FROM { 'имя_файла' | PROGRAM 'команда' | STDIN }
    [ [ WITH ] ( параметр [, ...] ) ]

COPY { имя_таблицы [ ( имя_столбца [, ...] ) ] | ( запрос ) }
    TO { 'имя_файла' | PROGRAM 'команда' | STDOUT }
    [ [ WITH ] ( параметр [, ...] ) ]

Здесь допускается параметр:

    FORMAT имя_формата
    OIDS [ boolean ]
    FREEZE [ boolean ]
    DELIMITER 'символ_разделитель'
    NULL 'маркер_NULL'
    HEADER [ boolean ]
    QUOTE 'символ_кавычек'
    ESCAPE 'символ_экранирования'
    FORCE_QUOTE { ( имя_столбца [, ...] ) | * }
    FORCE_NOT_NULL ( имя_столбца [, ...] )
    FORCE_NULL ( имя_столбца [, ...] )
    ENCODING 'имя_кодировки'
```

Описание

`COPY` перемещает данные между таблицами Postgres Pro и обычными файлами в файловой системе. `COPY TO` копирует содержимое таблицы *в* файл, а `COPY FROM` — *из* файла в таблицу (добавляет данные к тем, что уже содержались в таблице). `COPY TO` может также скопировать результаты запроса `SELECT`.

Если указывается список столбцов, `COPY TO` копирует в файл только данные указанных столбцов, а `COPY FROM` вставляет каждое поле из файла в соответствующий ему по порядку столбец из указанного списка. В случае отсутствия в этом списке каких-либо столбцов таблицы при `COPY FROM` они получают значения по умолчанию.

`COPY` с именем файла указывает серверу Postgres Pro читать или записывать непосредственно этот файл. Заданный файл должен быть доступен пользователю Postgres Pro (тому пользователю, от имени которого работает сервер), и путь к файлу должен задаваться с точки зрения сервера. Когда указывается параметр `PROGRAM`, сервер выполняет заданную команду и читает данные из стандартного вывода программы, либо записывает их в стандартный ввод. Команда должна определяться с точки зрения сервера и быть доступной для исполнения пользователю Postgres Pro. Когда указывается `STDIN` или `STDOUT`, данные передаются через соединение клиента с сервером.

Параметры

имя_таблицы

Имя существующей таблицы (возможно, дополненное схемой).

имя_столбца

Необязательный список столбцов, данные которых будут копироваться. Если этот список отсутствует, копируются все столбцы таблицы.

запрос

Команда SELECT, VALUES, INSERT, UPDATE или DELETE, результаты которой будут скопированы. Заметьте, что запрос должен заключаться в скобки.

Для запросов INSERT, UPDATE и DELETE должно задаваться предложение RETURNING и в целевом отношении не должно быть условного правила, правила ALSO или правила INSTEAD, разворачивающегося в несколько операторов.

имя_файла

Путь входного или выходного файла. Путь входного файла может быть абсолютным или относительным, но путь выходного должен быть только абсолютным. Пользователям Windows следует использовать формат E'' и продублировать каждую обратную черту в пути файла.

PROGRAM

Выполняемая команда. COPY FROM читает стандартный вывод команды, а COPY TO записывает в её стандартный ввод.

Заметьте, что команда запускается через командную оболочку, так что если требуется передать этой команде какие-либо аргументы, поступающие из недоверенного источника, необходимо аккуратно избавиться от всех спецсимволов, имеющих особое значение в оболочке, либо экранировать их. По соображениям безопасности лучше ограничиться фиксированной строкой команды или как минимум не позволять пользователям вводить в неё произвольное содержимое.

STDIN

Указывает, что данные будут поступать из клиентского приложения.

STDOUT

Указывает, что данные будут выдаваться клиентскому приложению.

boolean

Включает или отключает заданный параметр. Для включения параметра можно написать TRUE, ON или 1, а для отключения — FALSE, OFF или 0. Значение *boolean* можно опустить, в этом случае подразумевается TRUE.

FORMAT

Выбирает формат чтения или записи данных: text (текстовый), csv (значения, разделённые запятыми, Comma Separated Values) или binary (двоичный). По умолчанию выбирается формат text.

OIDS

Копирует OID каждой строки. (Если присутствует указание OIDS, но таблица не содержит столбец oid, либо копируется *запрос*, возникнет ошибка.)

FREEZE

Запросы копируют данные с уже замороженными строками, как после выполнения команды VACUUM FREEZE. Это позволяет увеличить производительность при начальном добавлении данных. Строки будут замораживаться, только если загружаемая таблица была создана или опустошена в текущей подтранзакции, с ней не связаны открытые курсоры и в данной транзакции нет других снимков. Выполнять COPY FREEZE с секционированной таблицей в настоящее время нельзя.

Заметьте, что все другие сеансы будут немедленно видеть данные, как только они будут успешно загружены. Это нарушает принятые правила видимости MVCC, так что пользователи, включающие этот режим, должны понимать, какие проблемы это может вызвать.

DELIMITER

Задаёт символ, разделяющий столбцы в строках файла. По умолчанию это символ табуляции в текстовом формате и запятая в формате CSV. Задаваемый символ должен быть однобайтовым. Для формата binary этот параметр не допускается.

NULL

Определяет строку, задающую значение NULL. По умолчанию в текстовом формате это \N (обратная косая черта и N), а в формате CSV — пустая строка без кавычек. Пустую строку можно использовать и в текстовом формате, если не требуется различать пустые строки и NULL. Для формата binary этот параметр не допускается.

> **Примечание**
>
> При выполнении COPY FROM любые значения, совпадающие с этой строкой, сохраняются как значение NULL, так что при переносе данных важно убедиться в том, что это та же строка, что применялась в COPY TO.

HEADER

Указывает, что файл содержит строку заголовка с именами столбцов. При выводе первая строка файла будет содержать имена столбцов таблицы, а при вводе первая строка просто игнорируется. Этот параметр допускается только для формата CSV.

QUOTE

Указывает символ кавычек, используемый для заключения данных в кавычки. По умолчанию это символ двойных кавычек. Задаваемый символ должен быть однобайтовым. Этот параметр поддерживается только для формата CSV.

ESCAPE

Задаёт символ, который будет выводиться перед символом данных, совпавшим со значением QUOTE. По умолчанию это тот же символ, что и QUOTE (то есть, при появлении в данных кавычек, они дублируются). Задаваемый символ должен быть однобайтовым. Этот параметр допускается только для режима CSV.

FORCE_QUOTE

Принудительно заключает в кавычки все значения не NULL в указанных столбцах. Выводимое значение NULL никогда не заключается в кавычки. Если указано *, в кавычки будут заключаться значения не NULL во всех столбцах. Этот параметр принимает только команда COPY TO и только для формата CSV.

FORCE_NOT_NULL

Не сопоставлять значения в указанных столбцах с маркером NULL. По умолчанию, когда маркер пуст, это означает, что пустые значения будут считаны как строки нулевой длины, а не NULL, даже когда они не заключены в кавычки. Этот параметр допускается только в команде COPY FROM и только для формата CSV.

FORCE_NULL

Сопоставлять значения в указанных столбцах с маркером NULL, даже если они заключены в кавычки, и в случае совпадения устанавливать значение NULL. По умолчанию, когда этот маркер пуст, пустая строка в кавычках будет преобразовываться в NULL. Этот параметр допускается только в команде COPY FROM и только для формата CSV.

ENCODING

Указывает, что файл имеет кодировку *имя_кодировки*. Если этот параметр опущен, выбирается текущая кодировка клиента. Подробнее об этом говорится ниже, в примечаниях.

Выводимая информация

В случае успешного завершения, COPY возвращает метку команды в виде

COPY *число*

Здесь *число* — количество скопированных записей.

> **Примечание**
>
> psql выводит эту метку, только если выполнялась не команда COPY ... TO STDOUT или её аналог в psql, метакоманда \copy ... to stdout. Это сделано для того, чтобы метка команды не смешалась с данными, выведенными перед ней.

Замечания

Команду COPY TO можно использовать только с простыми таблицами, не представлениями, и при этом она не копирует строки из дочерних таблиц или секций. То есть, COPY *таблица* TO копирует те же строки, что выдаёт запрос SELECT * FROM ONLY *таблица*. Для выгрузки всех строк представления или таблицы с учётом иерархии наследования или секционирования можно применить COPY (SELECT * FROM *таблица*) TO ....

COPY FROM можно применять с обычными таблицами и представлениями с триггерами INSTEAD OF INSERT.

В таблице, данные которой читает команда COPY TO, требуется иметь право на выборку данных, а в таблице, куда вставляет значения COPY FROM, требуется право на добавление. При этом, если в команде перечисляются избранные столбцы, достаточно иметь права только для них.

Если для таблицы включена защита на уровне строк, соответствующие политики SELECT будут применяться и к операторам COPY *таблица* TO. Операторы COPY FROM для таблиц с защитой строк в настоящее время не поддерживаются. Вместо них следует использовать равнозначные операторы INSERT.

Файлы, указанные в команде COPY, читаются или записываются непосредственно сервером, не клиентским приложением. Поэтому они должны располагаться на сервере или быть доступными серверу, а не клиенту. Они должны быть доступны на чтение или запись пользователю Postgres Pro (пользователю, от имени которого работает сервер), не клиенту. Аналогично, команда, указанная параметром PROGRAM, выполняется непосредственно сервером, а не клиентским приложением, и должна быть доступна на выполнение пользователю Postgres Pro. Выполнять команду COPY с файлом (или командой) разрешено только суперпользователям базы данных, так как она позволяет прочитать и записать любой файл, к которому имеет доступ сервер.

Не путайте команду COPY с реализованной в psql метакомандой \copy. Метакоманда \copy вызывает COPY FROM STDIN или COPY TO STDOUT, а затем работает с данными в файле, доступном клиенту psql. Таким образом, когда применяется команда \copy, доступность файла и права доступа зависят от клиента, а не от сервера.

Путь файла, указываемый в COPY, рекомендуется всегда задавать как абсолютный, а не относительный. Это обязательное условие для команды COPY TO, но COPY FROM позволяет прочитать файл, заданный и относительным путём. Такой путь будет интерпретироваться относительно рабочего каталога серверного процесса (обычно это каталог данных кластера), а не рабочего каталога клиента.

Выполнение команды в PROGRAM может быть ограничено и другими работающими в ОС механизмами контроля доступа, например SELinux.

COPY FROM вызывает все триггеры и обрабатывает все ограничения-проверки в целевой таблице. Однако правила при загрузке данных не вызываются.

Для столбцов идентификации команда COPY FROM всегда переносит значения, содержащиеся во входных данных, как команда INSERT с указанием OVERRIDING SYSTEM VALUE.

При вводе и выводе данных COPY учитывается DateStyle. Для обеспечения переносимости на другие инсталляции Postgres Pro, в которых могут использоваться нестандартные значения DateStyle, значение DateStyle следует установить равным ISO до вызова COPY TO. Также рекомендуется не выгружать данные с IntervalStyle равным sql_standard, так как сервер с другим значением IntervalStyle может неправильно воспринимать отрицательные интервалы в таких данных.

Входные данные интерпретируются согласно кодировке, заданной параметром ENCODING, или текущей кодировке клиента, а выходные кодируются в кодировке ENCODING или текущей кодировке клиента, даже если данные не проходят через клиента, а считываются или записываются в файл непосредственно сервером.

COPY прекращает операцию при первой ошибке. Это не должно приводить к проблемам в случае с COPY TO, но после COPY FROM в целевой таблице остаются ранее полученные строки. Эти строки не будут видимыми и доступными, но будут занимать место на диске. Если сбой происходит при копировании большого объёма данных, это может приводить к значительным потерям дискового пространства. При желании вернуть потерянный объём, это можно сделать с помощью команды VACUUM.

FORCE_NULL и FORCE_NOT_NULL можно применить одновременно к одному столбцу. В результате NULL-значения в кавычках будут преобразованы в NULL, а NULL-значения без кавычек — в пустые строки.

Postgres Pro не принимает нулевые байты в данных. Если вы хотите импортировать такие данные с помощью команды COPY FROM, вы можете определить в параметре конфигурации nul_byte_replacement_on_import ASCII-символ для замены нулевых байтов на лету.

Форматы файлов

Текстовый формат

Когда применяется формат text, читаемые или записываемые данные представляют собой текстовый файл, строка в котором соответствует строке таблицы. Столбцы в строке разделяются символом-разделителем. Значения самих столбцов — текстовые строки, выдаваемые функцией вывода, либо воспринимаемые функцией ввода, соответствующей типу данных столбца. Заданный маркер NULL выводится и считывается вместо столбцов со значением NULL. COPY FROM выдаёт ошибку, если в любой из строк во входном файле оказывается больше или меньше столбцов, чем ожидается. С указанием OIDS значение OID считывается или записывается в первом столбце, предшествующем столбцам с основными данными.

Конец данных может обозначаться одной строкой, содержащей только обратную косую и точку (\.). Маркер конца данных не требуется при чтении из файла, так как его роль вполне выполняет конец файла; он необходим только при передаче данных в/из клиентского приложения по протоколу обмена до версии 3.0.

Символы обратной косой черты (\) в данных COPY позволяют экранировать символы данных, которые без них считались бы разделителями строк или столбцов. В частности, предваряться обратной косой *должны* следующие символы, когда они оказываются в значении столбца: сама обратная косая черта, перевод строки, возврат каретки и текущий разделитель.

Маркер NULL передаётся команде COPY TO как есть, без добавления обратной косой; COPY FROM, со своей стороны, ищет во вводимых данных маркеры NULL до удаления обратных косых. Таким образом, маркер NULL, например такой как \N, отличается от значения \N в данных (оно должно представляться в виде \\N).

Команда COPY FROM распознаёт следующие спецпоследовательности:

Последовательность	**Представляет**
\b	Забой (ASCII 8)
\f	Подача формы (ASCII 12)
\n	Новая строка (ASCII 10)
\r	Возврат каретки (ASCII 13)
\t	Табуляция (ASCII 9)
\v	Вертикальная табуляция (ASCII 11)
цифры	Обратная косая с последующими 1–3 восьмеричными цифрами представляет байт с заданным числовым кодом
\x*цифры*	Обратная косая с последующим x и 1-2 шестнадцатеричными цифрами представляет байт с заданным числовым кодом

В настоящее время COPY TO никогда не выводит спецпоследовательности с восьмеричными или шестнадцатеричными кодами, однако выводит другие вышеперечисленные спецпоследовательности вместо управляющих символов.

Любой другой символ после обратной косой, отсутствующий в приведённой выше таблице, будет представлять себя. Однако опасайтесь излишнего добавления обратных косых, так как это может привести к случайному образованию строки, обозначающей маркер конца данных (\.) или маркер NULL (\N по умолчанию). Эти строки будут восприняты прежде, чем обработаются спецпоследовательности с обратной косой.

В приложениях, генерирующих данные для COPY, настоятельно рекомендуется преобразовать символы новой строки и возврата каретки в последовательности \n и \r, соответственно. В настоящее время можно представить возврат каретки в данных как обратная косая и возврат каретки, а перевод строки как обратная косая и перевод строки, однако это может не поддерживаться в будущих версиях. Такие символы также подвержены искажениям, если файл с выводом COPY переносится между разными системами (например, с Unix в Windows и наоборот).

Все последовательности с обратной косой чертой обрабатываются после преобразования кодировки. Байты, заданные в таких последовательностях восьмеричными или шестнадцатеричными цифрами, должны представлять допустимые символы в кодировке базы данных.

COPY TO завершает каждую строку символом новой строки в стиле Unix («\n»). Серверы, работающие в Microsoft Windows, вместо этого выводят символы возврат каретки/новая строка («\r \n»), но только при выводе COPY в файл на сервере; для согласованности на разных платформах, COPY TO STDOUT всегда передаёт «\n», вне зависимости от платформы сервера. COPY FROM может воспринимать строки, завершающиеся символами новая строка, перевод каретки, либо возврат каретки+новая строка. Чтобы уменьшить риск ошибки из-за неэкранированных символов новой строки и возврата каретки, которые должны были быть данными, COPY FROM сигнализирует о проблеме, если концы строк во входных данных различаются.

Формат CSV

Этот формат применяется для импорта и экспорта данных в виде списка значений, разделённых запятыми (CSV), с которым могут работать многие другие программы, например электронные таблицы. Вместо правил экранирования значений, введённых в Postgres Pro для текстового формата, этот формат использует стандартный механизм экранирования CSV.

Значения в каждой записи разделяются символами DELIMITER. Если значение содержит символ разделителя, символ QUOTE, маркер NULL, символ возврата каретки или перевода строки, то всё значение дополнятся спереди и сзади символами QUOTE, а любое вхождение символа QUOTE или

спецсимвола (ESCAPE) в данных предваряется спецсимволом. С указанием FORCE_QUOTE в кавычки будут принудительно заключаться любые значения не NULL в указанных столбцах.

В формате CSV отсутствует стандартный способ отличить значение NULL от пустой строки. В Postgres Pro команда COPY решает это с помощью кавычек. Значение NULL выводится в виде строки, задаваемой параметром NULL, и не заключается в кавычки, тогда как значение не NULL, со строкой, задаваемой параметром NULL, заключается. Например, с параметрами по умолчанию NULL записывается в виде пустой строки без кавычек, тогда как пустая строка записывается в двойных кавычках (""). При чтении значений действуют похожие правила. Указание FORCE_NOT_NULL позволяет избежать сравнений на NULL во входных данных в заданных столбцах, а FORCE_NULL — преобразовывать в NULL маркеры NULL, даже заключённые в кавычки.

Так как обратная косая черта не является спецсимволом в формате CSV, маркер конца данных \. может быть и значением данных. Во избежание ошибок интерпретации данные \., выводимые в виде единственного элемента строки, автоматически заключаются в кавычки при выводе, а при вводе этот маркер, заключённый в кавычки, не воспринимается как маркер конца данных. При загрузке файла, созданного другой программой, в котором в единственном столбце без кавычек оказалось значение \., потребуется дополнительно заключить это значение в кавычки.

Примечание

В формате CSV все символы являются значимыми. Заключённое в кавычки значение, дополненное пробелами или любыми другими символами, кроме DELIMITER, будет включать и эти символы. Это может приводить к ошибкам при импорте данных из системы, дополняющей строки CSV пробельными символами до некоторой фиксированной ширины. В случае возникновения такой проблемы необходимо обработать файл CSV и удалить из него замыкающие пробельные символы, прежде чем загружать данные из него в Postgres Pro.

Примечание

Обработчик формата CSV воспринимает и генерирует файлы CSV со значениями в кавычках, которые могут содержать символы возврата каретки и перевода строки. Таким образом, число строк в этих файлах не строго равно числу строк в таблице, как в файлах текстового формата.

Примечание

Многие программы генерируют странные и иногда неприемлемые файлы CSV, так что этот формат используется скорее по соглашению, чем по стандарту. Поэтому вам могут встретиться файлы, которые невозможно импортировать, используя этот механизм, а COPY может сформировать такие файлы, что их не смогут обработать другие программы.

Двоичный формат

При выборе формата binary все данные сохраняются/считываются в двоичном, а не текстовом виде. Иногда этот формат обрабатывается быстрее, чем текстовый и CSV, но он может оказаться непереносимым между разными машинными архитектурами и версиями Postgres Pro. Кроме того, двоичный формат сильно зависит от типов данных; например, он не позволяет вывести данные из столбца smallint, а затем прочитать их в столбец integer, хотя с текстовым форматом это вполне возможно.

Формат binary включает заголовок файла, ноль или более записей, содержащих данные строк, и окончание файла. Для заголовков и данных принят сетевой порядок байт.

> **Примечание**
>
> В PostgreSQL до версии 7.4 использовался другой двоичный формат.

Заголовок файла

Заголовок файла содержит 15 байт фиксированных полей, за которыми следует область расширения заголовка переменной длины. Фиксированные поля:

Сигнатура

Последовательность из 11 байт `PGCOPY\n\377\r\n\0` — заметьте, что нулевой байт является обязательной частью сигнатуры. (Эта сигнатура позволяет легко выявить файлы, испорченные при передаче, не сохраняющей все 8 бит данных. Она изменится при прохождении через фильтры, меняющие концы строк, отбрасывающие нулевые байты или старшие биты, либо добавляющие чётность.)

Поле флагов

Маска из 32 бит, обозначающая важные аспекты формата файла. Биты нумеруются от 0 (LSB) до 31 (MSB). Учтите, что это поле хранится в сетевом порядке байт (наиболее значащий байт первый), как и все целочисленные поля в этом формате. Биты 16-31 зарезервированы для обозначения критичных особенностей формата; обработчик должен прервать чтение, встретив любой неожиданный бит в этом диапазоне. Биты 0-15 зарезервированы для обозначения особенностей, связанных с обратной совместимостью; обработчик может просто игнорировать любые неожиданные биты в этом диапазоне. В настоящее время определён только один битовый флаг, остальные должны быть равны 0:

Бит 16

При 1 в данные включается OID; при 0 — нет

Длина области расширения заголовка

Целое 32-битное число, определяющее длину в байтах остального заголовка, не включая само это значение. В настоящее время содержит 0, и сразу за ним следует первая запись. При будущих изменениях формата в заголовок могут быть добавлены дополнительные данные. Обработчик должен просто пропускать все расширенные данные заголовка, о которых ему ничего не известно.

Область расширения заголовка предусмотрена для размещения последовательности самоопределяемых блоков. Поле флагов не должно содержать указаний о том, что содержится в области расширения. Точное содержимое области расширения может быть определено в будущих версиях.

При таком подходе возможно как обратно-совместимое дополнение заголовка (добавить блоки расширения заголовка или установить младшие биты флагов), так и не обратно-совместимое (установить старшие биты флагов, сигнализирующие о подобном изменении, и добавить вспомогательные данные в область расширения, если это потребуется).

Записи

Каждая запись начинается с 16-битного целого числа, определяющего количество полей в записи. (В настоящее время во всех записях должно быть одинаковое число полей, но так может быть не всегда.) Затем, для каждого поля в записи указывается 32-битная длина поля, за которой следует это количество байт с данными поля. (Значение длины не включает свой размер, и может быть равно нулю.) В качестве особого варианта, -1 обозначает, что в поле содержится NULL. В случае с NULL за длиной не следуют байты данных.

Выравнивание или какие-либо дополнительные данные между полями не вставляются.

В настоящее время предполагается, что все значения данных в файле двоичного формата содержатся в двоичном формате (формате под кодом 1). Возможно, в будущем расширении

в заголовок будет добавлено поле, позволяющее задавать другие коды форматов для разных столбцов.

Чтобы определить подходящий двоичный формат для фактических данных, обратитесь к исходному коду Postgres Pro, в частности, к функциям `*send` и `*recv` для типов данных каждого столбца (обычно эти функции находятся в каталоге `src/backend/utils/adt/` в дереве исходного кода).

Если в файл включается OID, поле OID следует немедленно за числом, определяющим количество полей. Это поле не отличается от других ничем, кроме того, что оно не учитывается в количестве полей. В частности, для него также задаётся длина — это позволяет обрабатывать и четырёх- и восьмибайтовые OID без особых сложностей, и даже вывести OID, равный NULL, если возникнет потребность в этом.

Окончание файла

Окончание файла состоит из 16-битного целого, содержащего -1. Это позволяет легко отличить его от счётчика полей в записи.

Обработчик, читающий файл, должен выдать ошибку, если число полей в записи не равно -1 или ожидаемому числу столбцов. Это обеспечивает дополнительную проверку синхронизации данных.

Примеры

В следующем примере таблица передаётся клиенту с разделителем полей «вертикальная черта» (`|`):

```
COPY country TO STDOUT (DELIMITER '|');
```

Копирование данных из файла в таблицу `country`:

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

Копирование в файл только данных стран, название которых начинается с 'A':

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO '/usr1/proj/bray/sql/
a_list_countries.copy';
```

Для копирования данных в сжатый файл можно направить вывод через внешнюю программу сжатия:

```
COPY country TO PROGRAM 'gzip > /usr1/proj/bray/sql/country_data.gz';
```

Пример данных, подходящих для копирования в таблицу из `STDIN`:

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
```

Примечание: пробелы в каждой строке на самом деле обозначают символы табуляции.

Ниже приведены те же данные, но выведенные в двоичном формате. Данные показаны после обработки Unix-утилитой `od -c`. Таблица содержит три столбца; первый имеет тип `char(2)`, второй — `text`, а третий — `integer`. Последний столбец во всех строках содержит NULL.

```
0000000   P   G   C   O   P   Y  \n 377  \r  \n  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0 003  \0  \0  \0 002   A   F  \0  \0  \0 013   A
0000040   F   G   H   A   N   I   S   T   A   N 377 377 377 377  \0 003
0000060  \0  \0  \0 002   A   L  \0  \0  \0 007   A   L   B   A   N   I
0000100   A 377 377 377 377  \0 003  \0  \0  \0 002   D   Z  \0  \0  \0
0000120 007   A   L   G   E   R   I   A 377 377 377 377  \0 003  \0  \0
0000140  \0 002   Z   M  \0  \0  \0 006   Z   A   M   B   I   A 377 377
```

```
0000160 377 377  \0 003  \0  \0  \0 002   Z   W  \0  \0  \0  \b   Z   I
0000200   M   B   A   B   W   E 377 377 377 377 377 377
```

Совместимость

Оператор `COPY` отсутствует в стандарте SQL.

До версии PostgreSQL 9.0 использовался и по-прежнему поддерживается следующий синтаксис:

```
COPY имя_таблицы [ ( имя_столбца [, ...] ) ]
    FROM { 'имя_файла' | STDIN }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'символ_разделитель' ]
          [ NULL [ AS ] 'маркер_NULL' ]
          [ CSV [ HEADER ]
                [ QUOTE [ AS ] 'символ_кавычек' ]
                [ ESCAPE [ AS ] 'символ_экранирования' ]
                [ FORCE NOT NULL имя_столбца [, ...] ] ] ]

COPY { имя_таблицы [ ( имя_столбца [, ...] ) ] | ( запрос ) }
    TO { 'имя_файла' | STDOUT }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'символ_разделитель' ]
          [ NULL [ AS ] 'маркер_NULL' ]
          [ CSV [ HEADER ]
                [ QUOTE [ AS ] 'символ_кавычек' ]
                [ ESCAPE [ AS ] 'символ_экранирования' ]
                [ FORCE QUOTE { имя_столбца [, ...] | * } ] ] ]
```

Заметьте, что в этом синтаксисе ключевые слова `BINARY` и `CSV` обрабатываются как независимые, а не как аргументы параметра `FORMAT`.

До версии PostgreSQL 7.3 использовался и по-прежнему поддерживается следующий синтаксис:

```
COPY [ BINARY ] имя_таблицы [ WITH OIDS ]
    FROM { 'имя_файла' | STDIN }
    [ [USING] DELIMITERS 'символ_разделитель' ]
    [ WITH NULL AS 'маркер_NULL' ]

COPY [ BINARY ] имя_таблицы [ WITH OIDS ]
    TO { 'имя_файла' | STDOUT }
    [ [USING] DELIMITERS 'символ_разделитель' ]
    [ WITH NULL AS 'маркер_NULL' ]
```

CREATE ACCESS METHOD

CREATE ACCESS METHOD — создать новый метод доступа

Синтаксис

```
CREATE ACCESS METHOD имя
    TYPE тип_метода_доступа
    HANDLER функция_обработчик
```

Описание

Команда `CREATE ACCESS METHOD` создаёт новый метод доступа.

Имя метода доступа должно быть уникальным в базе данных.

Определять новые методы доступа могут только суперпользователи.

Параметры

имя

Имя создаваемого метода доступа.

тип_метода_доступа

Это предложение задаёт тип создаваемого метода доступа. В настоящее время поддерживается только `INDEX`.

функция_обработчик

В аргументе *функция_обработчик* указывается имя (возможно, дополненное схемой) ранее зарегистрированной функции, представляющей метод доступа. Функция-обработчик должна принимать один аргумент типа `internal`, а тип её результата зависит от типа метода доступа; для методов доступа типа `INDEX` это должен быть `index_am_handler`. Также от типа метода доступа зависит API уровня C, который должна реализовывать эта функция-обработчик. API индексных методов доступа описан в Главе 56.

Примеры

Создание метода доступа индекса `heptree` с функцией-обработчиком `heptree_handler`:

```
CREATE ACCESS METHOD heptree TYPE INDEX HANDLER heptree_handler;
```

Совместимость

`CREATE ACCESS METHOD` является расширением Postgres Pro.

См. также

DROP ACCESS METHOD, CREATE OPERATOR CLASS, CREATE OPERATOR FAMILY

CREATE AGGREGATE

CREATE AGGREGATE — создать агрегатную функцию

Синтаксис

```
CREATE AGGREGATE имя ( [ режим_аргумента ] [ имя_аргумента ] тип_данных_аргумента
 [ , ... ] ) (
    SFUNC = функция_состояния,
    STYPE = тип_данных_состояния
    [ , SSPACE = размер_данных_состояния ]
    [ , FINALFUNC = функция_завершения ]
    [ , FINALFUNC_EXTRA ]
    [ , COMBINEFUNC = комбинирующая_функция ]
    [ , SERIALFUNC = функция_сериализации ]
    [ , DESERIALFUNC = функция_десериализации ]
    [ , INITCOND = начальное_условие ]
    [ , MSFUNC = функция_состояния_движ ]
    [ , MINVFUNC = обратная_функция_движ ]
    [ , MSTYPE = тип_данных_состояния_движ ]
    [ , MSSPACE = размер_данных_состояния_движ ]
    [ , MFINALFUNC = функция_завершения_движ ]
    [ , MFINALFUNC_EXTRA ]
    [ , MINITCOND = начальное_условие_движ ]
    [ , SORTOP = оператор_сортировки ]
    [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
)

CREATE AGGREGATE имя ( [ [ режим_аргумента ] [ имя_аргумента ] тип_данных_аргумента
 [ , ... ] ]
                        ORDER BY [ режим_аргумента ] [ имя_аргумента
 ] тип_данных_аргумента [ , ... ] ) (
    SFUNC = функция_состояния,
    STYPE = тип_данных_состояния
    [ , SSPACE = размер_данных_состояния ]
    [ , FINALFUNC = функция_завершения ]
    [ , FINALFUNC_EXTRA ]
    [ , INITCOND = начальное_условие ]
    [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
    [ , HYPOTHETICAL ]
)

или старый синтаксис

CREATE AGGREGATE имя (
    BASETYPE = базовый_тип,
    SFUNC = функция_состояния,
    STYPE = тип_данных_состояния
    [ , SSPACE = размер_данных_состояния ]
    [ , FINALFUNC = функция_завершения ]
    [ , FINALFUNC_EXTRA ]
    [ , COMBINEFUNC = комбинирующая_функция ]
    [ , SERIALFUNC = функция_сериализации ]
    [ , DESERIALFUNC = функция_десериализации ]
    [ , INITCOND = начальное_условие ]
    [ , MSFUNC = функция_состояния_движ ]
    [ , MINVFUNC = обратная_функция_движ ]
```

```
    [ , MSTYPE = тип_данных_состояния_движ ]
    [ , MSSPACE = размер_данных_состояния_движ ]
    [ , MFINALFUNC = функция_завершения_движ ]
    [ , MFINALFUNC_EXTRA ]
    [ , MINITCOND = начальное_условие_движ ]
    [ , SORTOP = оператор_сортировки ]
)
```

Описание

`CREATE AGGREGATE` создаёт новую агрегатную функцию. Некоторое количество базовых и часто используемых агрегатных функций включено в дистрибутив, они описаны в Разделе 9.20. Но если нужно адаптировать их к новым типам или создать недостающие агрегатные функции, это можно сделать с помощью команды `CREATE AGGREGATE`.

Если указывается имя схемы (например, `CREATE AGGREGATE myschema.myagg ...`), агрегатная функция создаётся в указанной схеме. В противном случае она создаётся в текущей схеме.

Агрегатная функция идентифицируется по имени и типам входных данных. Две агрегатных функции в одной схеме могут иметь одно имя, только если они работают с разными типами данных. Имя и тип(ы) входных данных агрегата не могут совпадать с именем и типами данных любой другой обычной функции в той же схеме. Это же правило действует при перегрузке имён обычных функций (см. CREATE FUNCTION).

Простую агрегатную функцию образуют одна или две обычные функции: функция перехода состояния *функция_состояния* и необязательная функция окончательного вычисления *функция_завершения*. Они используются следующим образом:

```
функция_состояния( внутреннее-состояние, следующие-значения-данных ) ---> следующее-
внутреннее-состояние
функция_завершения( внутреннее-состояние ) ---> агрегатное_значение
```

Postgres Pro создаёт временную переменную типа *тип_данных_состояния* для хранения текущего внутреннего состояния агрегата. Затем для каждой поступающей строки вычисляются значения аргументов агрегата и вызывается функция перехода состояния с текущим значением состояния и полученными аргументами; эта функция вычисляет следующее внутреннее состояние. Когда таким образом будут обработаны все строки, вызывается завершающая функция, которая должна вычислить возвращаемое значение агрегата. Если функция завершения отсутствует, просто возвращается конечное значение состояния.

Агрегатная функция может определить начальное условие, то есть начальное значение для внутренней переменной состояния. Это значение задаётся и сохраняется в базе данных в виде строки типа `text`, но оно должно быть допустимым внешним представлением константы типа данных переменной состояния. По умолчанию начальным значением состояния считается NULL.

Если функция перехода состояния объявлена как «strict» (строгая), её нельзя вызывать с входными значениями NULL. В этом случае агрегатная функция выполняется следующим образом. Строки со значениями NULL игнорируются (функция перехода не вызывается и предыдущее значение состояния не меняется) и если начальное состояние равно NULL, то в первой же строке, в которой все входные значения не NULL, первый аргумент заменяет значение состояния, а функция перехода вызывается для каждой последующей строки, в которой все входные значения не NULL. Это поведение удобно для реализации таких агрегатных функций, как `max`. Заметьте, что такое поведение возможно, только если *тип_данных_состояния* совпадает с первым *типом_данных_аргумента*. Если же эти типы различаются, необходимо задать начальное условие не NULL или использовать нестрогую функцию перехода состояния.

Если функция перехода состояния не является строгой, она вызывается безусловно для каждой поступающей строки и должна сама обрабатывать вводимые значения и переменную состояния, равные NULL. Это позволяет разработчику агрегатной функции полностью управлять тем, как она воспринимает значения NULL.

Если функция завершения объявлена как «strict» (строгая), она не будет вызвана при конечном значении состояния, равном NULL; вместо этого автоматически возвращается результат NULL. (Разумеется, это вполне нормальное поведение для строгих функций.) Когда функция завершения вызывается, она в любом случае может возвратить значение NULL. Например, функция завершения для `avg` возвращает NULL, если определяет, что было обработано ноль строк.

Иногда бывает полезно объявить функцию завершения как принимающую не только состояние, но и дополнительные параметры, соответствующие входным данным агрегата. В основном это имеет смысл для полиморфных функций завершения, которым может быть недостаточно знать тип данных только переменной состояния, чтобы вывести тип результата. Эти дополнительные параметры всегда передаются как NULL (так что функция завершения не должна быть строгой, когда применяется `FINALFUNC_EXTRA`), но в остальном это обычные параметры. Функция завершения может выяснить фактические типы аргументов в текущем вызове, воспользовавшись системным вызовом `get_fn_expr_argtype`.

Агрегатная функция может дополнительно поддерживать *режим движущегося агрегата*, как описано в Подразделе 35.10.1. Для этого режима требуются параметры `MSFUNC`, `MINVFUNC` и `MSTYPE`, а также могут задаваться `MSSPACE`, `MFINALFUNC`, `MFINALFUNC_EXTRA` и `MINITCOND`. За исключением `MINVFUNC`, эти параметры работают как соответствующие параметры простого агрегата без начальной буквы `M`; они определяют отдельную реализацию агрегата, включающую функцию обратного перехода.

Если в список параметров добавлено указание `ORDER BY`, создаётся особый типа агрегата, называемый *сортирующим агрегатом*; с указанием `HYPOTHETICAL` создаётся *гипотезирующий агрегат*. Эти агрегаты работают с группами отсортированных значений и зависят от порядка сортировки, поэтому определение порядка сортировки входных данных является неотъемлемой частью их вызова. Кроме того, они могут иметь *непосредственные* аргументы, которые вычисляются единожды для всей процедуры агрегирования, а не для каждой поступающей строки. Гипотезирующие агрегаты представляют собой подкласс сортирующих агрегатов, в которых непосредственные аргументы должны совпадать, по количеству и типам данных, с агрегируемыми аргументами. Это позволяет добавить значения этих непосредственных аргументов в набор агрегируемых строк в качестве дополнительной «гипотетической» строки.

Агрегатная функция может дополнительно поддерживать *частичное агрегирование*, как описано в Подразделе 35.10.4. Для этого требуется задать параметр `COMBINEFUNC`. Если в качестве *`типа_данных_состояния`* выбран `internal`, обычно уместно также задать `SERIALFUNC` и `DESERIALFUNC`, чтобы было возможно параллельное агрегирование. Заметьте, что для параллельного агрегирования агрегатная функция также должна быть помечена как `PARALLEL SAFE` (безопасная для распараллеливания).

Агрегаты, работающие подобно `MIN` и `MAX`, иногда можно соптимизировать, заменив сканирование всех строк таблицы обращением к индексу. Если агрегат подлежит такой оптимизации, это можно указать, определив *оператор сортировки*. Основное требование при этом: агрегат должен выдавать в результате первый элемент по порядку сортировки, задаваемому оператором; другими словами:

```
SELECT agg(col) FROM tab;
```

должно быть равнозначно:

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

Дополнительно предполагается, что агрегат игнорирует значения NULL и возвращает NULL, только если строк со значениями не NULL не нашлось. Обычно оператор `<` является подходящим оператором сортировки для `MIN`, а `>` — для `MAX`. Заметьте, что обращение к индексу может дать эффект, только если заданный оператор реализует стратегию «меньше» или «больше» в классе операторов индекса-B-дерева.

Чтобы создать агрегатную функцию, необходимо иметь право `USAGE` для типов аргументов, типа(ов) состояния и типа результата, а также право `EXECUTE` для опорных функций.

Параметры

имя

Имя создаваемой агрегатной функции (возможно, дополненное схемой).

режим_аргумента

Режим аргумента: `IN` или `VARIADIC`. (Агрегатные функции не поддерживают выходные аргументы (`OUT`).) По умолчанию подразумевается `IN`. Режим `VARIADIC` может быть указан только последним.

имя_аргумента

Имя аргумента. В настоящее время используется только в целях документирования. Если опущено, соответствующий аргумент будет безымянным.

тип_данных_аргумента

Тип входных данных, с которым работает эта агрегатная функция. Для создания агрегатной функции без аргументов вставьте `*` вместо списка с определениями аргументов. (Пример такой агрегатной функции: `count(*)`.)

базовый_тип

В прежнем синтаксисе `CREATE AGGREGATE` тип входных данных задавался параметром `basetype`, а не записывался после имени агрегата. Это позволяло указать только один входной параметр. Чтобы определить функцию без аргументов, используя этот синтаксис, в качестве значения `basetype` нужно указать `"ANY"` (не `*`). Создать сортирующий агрегат старый синтаксис не позволял.

функция_состояния

Имя функции перехода состояния, вызываемой для каждой входной строки. Для обычных агрегатных функций с *N* аргументами, *функция_состояния* должна принимать *N*+1 аргумент, первый должен иметь тип *тип_данных_состояния*, а остальные — типы соответствующих входных данных. Возвращать она должна значение типа *тип_данных_состояния*. Эта функция принимает текущее значение состояния и текущие значения входных данных, и возвращает следующее значение состояния.

В сортирующих (и в том числе, гипотезирующих) агрегатах функция перехода состояния получает только текущее значение состояния и агрегируемые аргументы, без непосредственных аргументов. Других отличий у неё нет.

тип_данных_состояния

Тип данных значения состояния для агрегатной функции.

размер_данных_состояния

Средний размер значения состояния агрегата (в байтах). Если этот параметр опущен или равен нулю, применяемая оценка по умолчанию определяется по типу *тип_данных_состояния*. Планировщик использует это значение для оценивания объёма памяти, требуемого для агрегатного запроса с группировкой. Планировщик может применить агрегирование по хешу для такого запроса, только если хеш-таблица, судя по оценке, поместится в work_mem; таким образом, при больших значениях этого параметра агрегирование по хешу будет менее желательным.

функция_завершения

Имя функции завершения, вызываемой для вычисления результата агрегатной функции после обработки всех входных строк. Для обычного агрегата эта функция должна принимать единственный аргумент типа *тип_данных_состояния*. Возвращаемым типом агрегата будет тип, который возвращает эта функция. Если *функция_завершения* не указана, результатом агрегата будет конечное значение состояния, а типом результата — *тип_данных_состояния*.

В сортирующих (и в том числе, гипотезирующих) агрегатах функция завершения получает не только конечное значение состояния, но и значения всех непосредственных аргументов.

Если команда содержит указание FINALFUNC_EXTRA, то в дополнение к конечному значению состояния и всем непосредственным аргументам функция завершения получает добавочные значения NULL, соответствующие обычным (агрегируемым) аргументам агрегата. Это в основном полезно для правильного определения типа результата при создании полиморфной агрегатной функции.

комбинирующая_функция

Дополнительно может быть указана *комбинирующая_функция*, чтобы агрегатная функция поддерживала частичное агрегирование. Если задаётся, *комбинирующая_функция* должна комбинировать два значения *типа_данных_состояния*, содержащих результат агрегирования по некоторому подмножеству входных значений, и вычислять новое значение *типа_данных_состояния*, представляющее результат агрегирования по обоим множествам данных. Эту функцию можно считать своего рода *функцией_состояния*, которая вместо обработки отдельной входной строки и включения её данных в текущее агрегируемое состояние включает некоторое агрегированное состояние в текущее.

Указанная *комбинирующая_функция* должна быть объявлена как принимающая два аргумента *типа_данных_состояния* и возвращающая значение *типа_данных_состояния*. Эта функция дополнительно может быть объявлена «строгой». В этом случае данная функция не будет вызываться, когда одно из входных состояний — NULL; в качестве корректного результата будет выдано другое состояние.

Для агрегатных функций, у которых *тип_данных_состояния* — internal, *комбинирующая_функция* не должна быть «строгой». При этом *комбинирующая_функция* должна позаботиться о том, чтобы состояния NULL обрабатывались корректно и возвращаемое состояние располагалось в контексте памяти агрегирования.

функция_сериализации

Агрегатная функция, у которой *тип_данных_состояния* — internal, может участвовать в параллельном агрегировании, только если для неё задана *функция_сериализации*, которая должна сериализовать агрегатное состояние в значение bytea для передачи другому процессу. Эта функция должна принимать один аргумент типа internal и возвращать тип bytea. Также при этом нужно задать соответствующую *функцию_десериализации*.

функция_десериализации

Десериализует ранее сериализованное агрегатное состояние обратно в *тип_данных_состояния*. Эта функция должна принимать два аргумента типов bytea и internal и выдавать результат типа internal. (Замечание: второй аргумент типа internal не используется, но требуется из соображений типобезопасности.)

начальное_условие

Начальное значение переменной состояния. Оно должно задаваться строковой константой в форме, пригодной для ввода в *тип_данных_состояния*. Если не указано, начальным значением состояния будет NULL.

функция_состояния_движ

Имя функции прямого перехода состояния, вызываемой для каждой входной строки в режиме движущегося агрегата. Это точно такая же функция, как и обычная функция перехода, но её первый аргумент и результат имеют тип *тип_данных_состояния_движ*, который может отличаться от типа *тип_данных_состояния*.

обратная_функция_движ

Имя функции обратного перехода состояния, применяемой в режиме движущегося агрегата. У этой функции те же типы аргумента и результатов, что и у *функции_состояния_движ*, но она

предназначена не для добавления, а для удаления значения из текущего состояния агрегата. Функция обратного перехода должна иметь ту же характеристику строгости, что и функция прямого перехода.

тип_данных_состояния_движ

Тип данных значения состояния для агрегатной функции в режиме движущегося агрегата.

размер_данных_состояния_движ

Примерный размер значения состояния в режиме движущегося агрегата. Он имеет то же значение, что и *размер_данных_состояния*.

функция_завершения_движ

Имя функции завершения, вызываемой в режиме движущегося агрегата для вычисления результата агрегатной функции после обработки всех входных строк. Она работает так же, как *функция_завершения*, но её первый аргумент имеет тип *тип_данных_состояния_движ*, а дополнительными пустыми аргументами управляет параметр MFINALFUNC_EXTRA. Тип результата, который определяет *функция_завершения_движ*, или *тип_данных_состояния_движ*, должен совпадать с типом результата обычной реализации агрегата.

начальное_условие_движ

Начальное значение переменной состояния в режиме движущегося агрегата. Оно применяется так же, как *начальное_условие*.

оператор_сортировки

Связанный оператор сортировки для реализации агрегатов, подобных MIN или MAX. Здесь указывается просто имя оператора (возможно, дополненное схемой). Предполагается, что оператор поддерживает те же типы входных данных, что и агрегат (который должен быть обычным и иметь один аргумент).

PARALLEL

Указания PARALLEL SAFE, PARALLEL RESTRICTED и PARALLEL UNSAFE имеют те же значения, что и в CREATE FUNCTION. Агрегатная функция не будет считаться распараллеливаемой, если она имеет характеристику PARALLEL UNSAFE (она подразумевается по умолчанию!) или PARALLEL RESTRICTED. Заметьте, что планировщик не обращает внимание на допустимость распараллеливания опорных функций агрегата, а учитывает только характеристику самой агрегатной функции.

HYPOTHETICAL

Этот признак, допустимый только для сортирующих агрегатов, указывает, что агрегатные аргументы должны обрабатываться согласно требованиям гипотезирующих агрегатов: то есть последние несколько непосредственных аргументов должны соответствовать по типам агрегатным аргументам (WITHIN GROUP). Признак HYPOTHETICAL не влияет на поведение во время выполнения, он учитывается только при разрешении типов данных и правил сортировки аргументов.

Параметры CREATE AGGREGATE могут записываться в любом порядке, не обязательно так, как показано выше.

Замечания

В параметрах, определяющих имена вспомогательных функций, при необходимости можно написать имя схемы, например: SFUNC = public.sum. Однако типы аргументов там не указываются — типы аргументов вспомогательных функций определяются другими параметрами.

Если агрегатная функция поддерживает режим движущегося агрегата, это увеличивает эффективность вычислений, когда она применяется в качестве оконной функции для окна с

движущимся началом рамки (то есть когда начало определяется не как UNBOUNDED PRECEDING). По сути, функция прямого перехода добавляет входные значения к состоянию агрегата, когда они поступают в рамку окна снизу, а функция обратного перехода снова вычитает их, когда они покидают рамку сверху. Поэтому вычитаются значения в том же порядке, в каком добавлялись. Когда бы ни вызывалась функция обратного перехода, она таким образом получит первое из добавленных, но ещё не удалённых значений аргумента. Функция обратного перехода может рассчитывать на то, что после того, как она удалит самые старые данные, в текущем состоянии останется ещё как минимум одна строка. (Когда это правило могло бы нарушиться, механизм оконных функций просто начинает агрегировать данные заново, а не вызывает функцию обратного перехода.)

Функция прямого перехода для режима движущегося агрегата не может возвращать NULL в качестве нового значения состояния. Если NULL возвращает функция обратного перехода, это показывает, что она не может произвести обратное вычисление для этих конкретных данных, и что вычисление агрегата следует выполнить заново от начальной позиции текущей рамки. Благодаря этому соглашению, режим движущегося агрегата можно использовать, даже если иногда возникают ситуации, в которых обратный расчёт состояния производить непрактично.

Агрегатную функцию можно использовать с движущимися рамками и без реализации движущегося агрегата, но при этом Postgres Pro будет заново агрегировать все данные при каждом перемещении начала рамки. Заметьте, что вне зависимости от того, поддерживает ли агрегатная функция режим движущегося агрегата, Postgres Pro может обойтись без повторных вычислений при сдвиге конца рамки; новые значения просто продолжают добавляться в состояние агрегата. Предполагается, что функция завершения не повреждает значение состояния агрегата, так что вычисление агрегата можно продолжить даже после получения результата для строк в определённой рамке.

Синтаксис сортирующих агрегатных функций позволяет указать VARIADIC и в последнем непосредственном параметре, и в последнем агрегатном (WITHIN GROUP). Однако в текущей реализации на применение VARIADIC накладываются два ограничения. Во-первых, в сортирующих агрегатах можно использовать только VARIADIC "any", но не другие типы переменных массивов. Во-вторых, если последним непосредственным аргументом является VARIADIC "any", то допускается только один агрегатный аргумент и это тоже должен быть VARIADIC "any". (В представлении, используемом в системных каталогах, эти два параметра объединяются в один элемент VARIADIC "any", так как в pg_proc нельзя представить функцию с несколькими параметрами VARIADIC.) Если агрегатная функция является гипотезирующей, непосредственные аргументы, соответствующие параметру VARIADIC "any", будут гипотетическими; любые предшествующие параметры представляют дополнительные непосредственные аргументы, которые могут не соответствовать агрегатным.

В настоящее время сортирующие агрегатные функции не поддерживают режим движущегося агрегата, так как их нельзя применять в качестве оконных функций.

Частичное (в том числе, параллельное) агрегирование в настоящее время не поддерживается для сортирующих агрегатных функций. Также оно никогда не будет применяться для агрегатных вызовов с предложениями DISTINCT или ORDER BY, так как они по природе своей не могут быть реализованы с частичным агрегированием.

Примеры

См. Раздел 35.10.

Совместимость

Оператор CREATE AGGREGATE является языковым расширением Postgres Pro. В стандарте SQL не предусмотрено создание пользовательских агрегатных функций.

См. также

ALTER AGGREGATE, DROP AGGREGATE

CREATE CAST

CREATE CAST — создать приведение

Синтаксис

```
CREATE CAST (исходный_тип AS целевой_тип)
    WITH FUNCTION имя_функции [ (тип_аргумента [, ...]) ]
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (исходный_тип AS целевой_тип)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (исходный_тип AS целевой_тип)
    WITH INOUT
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

Описание

`CREATE CAST` создаёт новое приведение. Приведение определяет, как выполнить преобразование из одного типа в другой. Например,

```
SELECT CAST(42 AS float8);
```

преобразует целочисленную константу 42 к типу `float8`, вызывая ранее определённую функцию, в данном случае `float8(int4)`. (Если подходящее приведение не определено, возникнет ошибка преобразования.)

Два типа могут быть *двоично-сводимыми*; это означает, что преобразование может быть выполнено «бесплатно», без вызова какой-либо функции. Для этого требуется, чтобы соответствующие значения имели одинаковое внутреннее представление. Например, типы `text` и `varchar` двоично-сводимые в обе стороны. Отношение двоичной сводимости не обязательно симметрично. Например, приведение типа `xml` к типу `text` в текущей реализации можно выполнить бесплатно, но для преобразования в обратном направлении требуется функция, выполняющая как минимум синтаксическую проверку. (Два типа, двоично-сводимые в обе стороны, также называются двоично-совместимыми.)

Приведение можно определить как *преобразование ввода/вывода*, используя указание `WITH INOUT`. В этом случае для приведения одного типа к другому вызывается функция вывода исходного типа данных, а выданная ей строка передаётся функции ввода целевого типа. Во многих случаях эта возможность избавляет от необходимости писать для преобразования всех типов отдельные функции приведения. Преобразование ввода/вывода работает так же, как и обычное приведение с функцией; отличается только реализация.

По умолчанию, приведение можно вызвать, только записав его явно, то есть применив конструкцию `CAST(`*x* `AS` *имя_типа*`)` или *x*`::`*имя_типа*.

Если приведение помечено `AS ASSIGNMENT`, его можно вызывать неявно, присваивая значение столбцу с целевым типом данных. Например, если `foo.f1` — столбец типа `text`, то команда:

```
INSERT INTO foo (f1) VALUES (42);
```

будет допустимой, если приведение типа `integer` к `text` помечено `AS ASSIGNMENT`, и не будет в противном случае. (Для описания такого типа приведений мы обычно используем термин *приведение присваивания*.)

Если приведение помечено `AS IMPLICIT`, оно будет вызываться неявно в любом контексте, будь то присваивание или внутреннее преобразование в выражении. (Обычно мы называет приведение такого типа *неявным приведением*.) Например, рассмотрите этот запрос:

```
SELECT 2 + 4.0;
```

При разборе запроса константам сначала назначаются типы `integer` и `numeric`. Однако в системных каталогах нет оператора `integer + numeric`, хотя есть оператор `numeric + numeric`. Таким образом, запрос выполнится успешно, если существует преобразование типа `integer` к `numeric` с пометкой `AS IMPLICIT` — и на самом деле это так. Анализатор запроса применит неявное приведение и запрос будет обработан, как если бы он был записан в виде

```
SELECT CAST ( 2 AS numeric ) + 4.0;
```

Системные каталоги также содержат приведение типа `numeric` к `integer`. Если бы это приведение тоже было бы помечено `AS IMPLICIT` (на самом деле это не так), анализатору запроса пришлось бы выбирать между предыдущим вариантом и приведением константы `numeric` к типу `integer` с последующим применением оператора `integer + integer`. Не имея возможности выбрать лучший вариант, анализатор бы не смог разрешить запрос и объявил бы его неоднозначным. Именно благодаря тому, что только одно из двух приведений сделано неявным, анализатор приходит к пониманию, что предпочитаемым является преобразование выражения `numeric`-и-`integer` в `numeric`; отдельного встроенного знания об этом нет.

Определяя, объявлять ли приведения неявными, разумно проявлять консерватизм. При чрезмерном количестве способов неявного приведения Postgres Pro может выбирать неожиданные интерпретации команд, или вовсе не сможет выполнить команды из-за наличия множества возможных интерпретаций. Как правило, следует делать приведение неявно вызываемым только для преобразований, сохраняющих информацию, между типами в одной общей категории типов. Например, приведение `int2` к `int4` разумно сделать неявным, но приведение `float8` к `int4`, возможно, лучше сделать только приведением присваивания. Приведения типов разных категорий, например, `text` к `int4`, лучше делать только явными.

> **Примечание**
>
> Иногда ради удобства или соответствия стандартам требуется ввести множество неявных преобразований для нескольких типов, что приводит к неизбежной неоднозначности. Чтобы анализатор запроса мог обеспечить желаемое поведение в таких случаях, он дополнительно принимает во внимание *категории типов* и *предпочитаемые типы*. Подробнее это описано в CREATE TYPE.

Чтобы создать приведение, необходимо быть владельцем одного (исходного или целевого) типа и иметь право `USAGE` для другого типа. Создать двоично-сводимое приведение могут только суперпользователи. (Это ограничение введено потому, что преобразование данных с ошибочным двоичным сведением может легко вызывать сбой сервера.)

Параметры

исходный_тип

Имя исходного типа данных для приведения.

целевой_тип

Имя целевого типа данных для приведения.

имя_функции[(*тип_аргумента* [, ...])]

Функция, вызываемая для выполнения приведения. Имя функции может быть дополнено схемой; в противном случае для поиска функции просматривается путь поиска. Тип данных результата должен соответствовать целевому типу приведения. Аргументы функции рассматриваются ниже. Если список аргументов отсутствует, имя функции должно быть уникальным в её схеме.

`WITHOUT FUNCTION`

Обозначает, что исходный тип сводится к целевому на двоичном уровне, так что функция для приведения не требуется.

`WITH INOUT`

Обозначает, что приведение выполняется как преобразование ввода/вывода, то есть вызывается функция вывода исходного типа данных, а её результат-строка передаётся функции ввода целевого типа.

`AS ASSIGNMENT`

Обозначает, что приведение может вызываться неявно в контексте присваивания.

`AS IMPLICIT`

Обозначает, что приведение может вызываться неявно в любом контексте.

Функции, реализующие приведение, могут иметь от одного до трёх аргументов. Тип первого аргумента должен быть идентичен или двоично-сводимым к исходному типу приведения. Второй аргумент, если он есть, должен иметь тип `integer`; в нём передаётся модификатор типа, связанный с целевым типом, или `-1`, если он отсутствует. Третий аргумент, если он есть, должен иметь тип `boolean`; в нём передаётся `true`, если приведение выполняется явно, либо `false` в противном случае. (Это довольно экстравагантно, но стандарт SQL предусматривает разное поведение для явного и неявного приведения в некоторых случаях. Этот аргумент предназначен для функций, которые должны реализовывать такие приведения. Однако создавать собственные типы данных, для которых это имело бы значение, не рекомендуется.)

Возвращаемый тип функции приведения должен быть идентичным или двоично-сводимым к целевому типу приведения.

Обычно исходный и целевой типы в приведении различаются, однако можно объявить приведение одного типа к такому же, если функция, реализующая преобразование, имеет более одного аргумента. Это используется для представления в системных каталогах функций, сводящих разные длины типов. Реализующая такое приведение функция будет сводить значение типа к значению с определённым модификатором, заданному вторым аргументом.

Когда исходный и целевой типы приведения различаются и функция принимает более одного аргумента, преобразование типа из одного в другой и сведение к нужной длине может выполняться за один шаг. Если же соответствующей записи не находится, приведение к типу с определённым модификатором выполняется в два этапа: сначала выполняется преобразование типа, а затем применяется модификатор типа.

Приведение типа домена или к типу домена в настоящее время не осуществляется. При попытке выполнить такое приведение вместо него выполняется приведение, связанное с базовым типом домена.

Замечания

Для удаления приведений, созданных пользователем, применяется DROP CAST.

Помните, что когда требуется преобразовывать типы в обе стороны, необходимо явно описать два приведения.

Обычно не требуется создавать приведения между пользовательскими типами и стандартными строковыми типами (`text`, `varchar` и `char(n)`, а также пользовательскими типами, относящимися к категории строковых). Для них Postgres Pro предоставляет автоматическое преобразование ввода/вывода. Автоматические приведения к строковым типам считаются приведениями присваивания, а автоматические приведения строковых типов к другим могут быть только явными. Это поведение можно переопределить, создав собственное приведение, заменяющее автоматическое, но обычно

это нужно, только чтобы сделать вызов более удобным, чем стандартное только присваивание или явное указание. Возможен и другой повод для такого переопределения — желание создать приведение, работающее не так, как функция ввода/вывода типа; но это настолько удивительно, что следует дважды подумать, хороша ли эта идея. (На самом деле у небольшого количества встроенных типов имеются подобные специфические приведения, в основном из-за требований стандарта SQL.)

Хотя это и не обязательно, но рекомендуется следовать старому соглашению называть функции, реализующие приведение, по целевому типу данных. Многие привыкли выполнять преобразование типов данных, записывая его в стиле функций, т. е. *имя_типа*(*x*). Эта запись на самом деле ни больше ни меньше как просто вызов функции, реализующей приведение; такой вызов не воспринимается как именно приведение. Если называть функции, не следуя этому соглашению, это может оказаться неожиданным для пользователей. Так как Postgres Pro позволяет перегружать одно и то же имя функции с разными типами аргументов, ничто не мешает создать множество функций приведения разных типов к одному, названных по имени этого целевого типа.

Примечание

Вообще говоря, в предыдущем абзаце допущено некоторое упрощение: есть два случая, когда конструкция с вызовом функции исполняется как приведение, без сопоставления с фактической функцией. Если вызову функции *имя*(*x*) в точности не соответствует существующая функция, но имеется тип данных *имя* и в `pg_cast` есть двоично-сводимое приведение типа *x* к этому типу, такой вызов будет воспринят как приведение. Это исключение введено, чтобы двоично-сводимое приведение можно было вызывать, используя синтаксис функций, несмотря на то, что никакой функции преобразования у него нет. Аналогично, если запись приведения в `pg_cast` отсутствует, но в случае приведения это было бы преобразование в/из строкового типа, такой вызов будет выполнен как преобразование ввода/вывода. Это исключение позволяет вызывать преобразование ввода/вывода, используя синтаксис вызова функции.

Примечание

Но есть исключение и из этого исключения: преобразование ввода/вывода из составных типов в строковые нельзя вызвать в виде функции, его необходимо записать как явное приведение (используя `CAST` или запись `::`). Это исключение было добавлено, потому что после введения автоматически предоставляемых преобразований ввода/вывода, оказалось слишком легко случайно вызвать такое приведение, тогда как имелась в виду ссылка на столбец или функцию.

Примеры

Создание приведения присваивания типа `bigint` к типу `int4` с помощью функции `int4(bigint)`:

```
CREATE CAST (bigint AS int4) WITH FUNCTION int4(bigint) AS ASSIGNMENT;
```

(Это приведение уже предопределено в системе.)

Совместимость

Команда `CREATE CAST` соответствует стандарту SQL, за исключением того, что в стандарте ничего не говорится о двоично-сводимых типах и дополнительных аргументах реализующих функций. Указание `AS IMPLICIT` тоже является расширением Postgres Pro.

См. также

CREATE FUNCTION, CREATE TYPE, DROP CAST

CREATE COLLATION

CREATE COLLATION — создать правило сортировки

Синтаксис

```
CREATE COLLATION [ IF NOT EXISTS ] имя (
    [ LOCALE = локаль, ]
    [ LC_COLLATE = категория_сортировки, ]
    [ LC_CTYPE = категория_типов_символов, ]
    [ PROVIDER = провайдер, ]
    [ VERSION = версия ]
)
CREATE COLLATION [ IF NOT EXISTS ] имя FROM существующее_правило
```

Описание

`CREATE COLLATION` определяет новое правило сортировки, используя параметры локали операционной системы, либо копируя существующее правило.

Чтобы создать правило сортировки, необходимо иметь право `CREATE` в целевой схеме.

Параметры

`IF NOT EXISTS`

Не считать ошибкой, если правило сортировки с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее правило сортировки как-то соотносится с тем, которое могло бы быть создано.

имя

Имя правила сортировки, возможно, дополненное схемой. Если схема не указана, правило сортировки создаётся в текущей схеме. Заданное имя правила должно быть уникальным в этой схеме. (Системные каталоги могут содержать правила сортировки с одним именем, но предназначенные для разных кодировок, однако они будут игнорироваться, если их кодировка не совпадает с кодировкой базы данных.)

локаль

Это краткая запись для одновременной установки `LC_COLLATE` и `LC_CTYPE`. Если указан этот вариант, задать любой из этих параметров отдельно нельзя.

категория_сортировки

Указанная локаль операционной системы устанавливается в качестве категории локали `LC_COLLATE`.

категория_типов_символов

Указанная локаль операционной системы устанавливается в качестве категории локали `LC_CTYPE`.

провайдер

Задаёт провайдер, который будет использоваться для функций локализации, связанных с данным правилом сортировки. Возможные значения: `icu`, `libc`. По умолчанию выбирается `libc`. Набор доступных значений зависит от операционной системы и параметров сборки.

версия

Задаёт строку версии, сохраняемую с правилом сортировки. Обычно её не следует задавать — тогда эта версия будет получена из фактической версии правила сортировки, сообщённой

операционной системой. Это указание предназначено для того, чтобы команда `pg_upgrade` смогла скопировать версию из существующей инсталляции.

Что делать при несовпадении версий правил сортировки, описано в ALTER COLLATION.

существующее_правило

Имя копируемого существующего правила сортировки. Новое правило сортировки получит те же свойства, что и существующее, но будет независимым объектом.

Замечания

Для удаления созданных пользователем правил сортировки применяется команда `DROP COLLATION`.

О поддержке правил сортировки в Postgres Pro рассказывается в Разделе 22.2. Дополнительные сведения о создании правил сортировки можно найти в Подразделе 22.2.2.3.

Когда используется провайдер `libc`, локаль должна быть применимой к кодировке текущей базы данных. Точные правила описаны в CREATE DATABASE.

Примеры

Создание правила сортировки из локали операционной системы `ru_RU.utf8` (предполагается, что кодировка текущей базы данных — `UTF8`):

```
CREATE COLLATION russian (locale = 'ru_RU.utf8');
```

Создание правила сортировки с порядком, при котором латинские буквы идут перед буквами кириллицы, с использованием провайдера ICU:

```
CREATE COLLATION latn_cyrl (provider = icu, locale = 'ru-RU-u-kr-latn-cyrl');
```

Создание правила сортировки из уже существующего:

```
CREATE COLLATION german FROM "de_DE";
```

Иногда удобно использовать в приложениях имена правил сортировки, не зависящие от операционной системы.

Совместимость

Оператор `CREATE COLLATION` определён в стандарте SQL, но его действие ограничено копированием существующего правила сортировки. Синтаксис создания нового правила сортировки представляет собой расширение Postgres Pro.

См. также

ALTER COLLATION, DROP COLLATION

CREATE CONVERSION

CREATE CONVERSION — создать перекодировку

Синтаксис

```
CREATE [ DEFAULT ] CONVERSION имя
    FOR исходная_кодировка TO целевая_кодировка FROM имя_функции
```

Описание

`CREATE CONVERSION` определяет новую перекодировку наборов символов. Кроме того, перекодировки, помеченные как `DEFAULT`, могут применяться для автоматического преобразования кодировки между клиентом и сервером. Для такого применения должны быть определены две перекодировки: из кодировки A в B *и* из кодировки B в A.

Чтобы создать перекодировку, необходимо иметь право `EXECUTE` для реализующей функции и право `CREATE` в целевой схеме.

Параметры

`DEFAULT`

Предложение `DEFAULT` показывает, что эта перекодировка должна использоваться по умолчанию для преобразования заданной исходной кодировки в целевую. Для каждой пары кодировок может быть только одна перекодировка по умолчанию.

имя

Имя перекодировки, возможно, дополненное схемой. Если схема не указана, перекодировка создаётся в текущей схеме. Имя перекодировки должно быть уникально в этой схеме.

исходная_кодировка

Имя исходной кодировки.

целевая_кодировка

Имя целевой кодировки.

имя_функции

Функция, выполняющая перекодирование. Имя функции может быть дополнено схемой, в противном случае для поиска функции просматривается путь поиска.

Функция должна иметь следующую сигнатуру:

```
conv_proc(
    integer,  -- идентификатор исходной кодировки
    integer,  -- идентификатор целевой кодировки
    cstring,  -- исходная строка (строка, завершающаяся 0, как в C)
    internal, -- целевая строка (заполняется строкой, завершающейся 0, как в C)
    integer   -- длина исходной строки
) RETURNS void;
```

Замечания

Для удаления перекодировок, созданных пользователем, применяется `DROP CONVERSION`.

Набор прав, требуемых для создания перекодировки, может измениться в будущих версиях.

Примеры

Создание перекодировки из кодировки `UTF8` в `LATIN1` с использованием функции `myfunc`:

```
CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc;
```

Совместимость

Оператор `CREATE CONVERSION` является расширением Postgres Pro. В стандарте SQL отсутствует оператор `CREATE CONVERSION`, но есть очень похожий по предназначению и синтаксису оператор `CREATE TRANSLATION`.

См. также

ALTER CONVERSION, CREATE FUNCTION, DROP CONVERSION

CREATE DATABASE

CREATE DATABASE — создать базу данных

Синтаксис

```
CREATE DATABASE имя
    [ WITH ] [ OWNER [=] имя_пользователя ]
           [ TEMPLATE [=] шаблон ]
           [ ENCODING [=] кодировка ]
           [ LC_COLLATE [=] категория_сортировки[@провайдер] ]
           [ LC_CTYPE [=] категория_типов_символов ]
           [ TABLESPACE [=] табл_пространство ]
           [ ALLOW_CONNECTIONS [=] разр_подключения ]
           [ CONNECTION LIMIT [=] предел_подключений ]
           [ IS_TEMPLATE [=] это_шаблон ]
```

Описание

Команда `CREATE DATABASE` создаёт базу данных Postgres Pro.

Чтобы создать базу данных, необходимо быть суперпользователем или иметь специальное право `CREATEDB`. См. CREATE USER.

По умолчанию новая база данных создаётся копированием стандартной системной базы данных `template1`. Задать другой шаблон можно, добавив указание `TEMPLATE` *`имя`*. В частности, написав `TEMPLATE template0`, можно создать девственно чистую базу данных, содержащую только стандартные объекты, предопределённые установленной версией Postgres Pro. Это бывает полезно, когда копировать в новую базу любые дополнительные объекты, добавленные локально в `template1`, нежелательно.

Параметры

`имя`

Имя создаваемой базы данных.

`имя_пользователя`

Имя пользователя (роли), назначаемого владельцем новой базы данных, либо `DEFAULT`, чтобы владельцем стал пользователь по умолчанию (а именно, пользователь, выполняющий команду). Чтобы создать базу данных и сделать её владельцем другую роль, необходимо быть непосредственным или опосредованным членом этой роли, либо суперпользователем.

`шаблон`

Имя шаблона, из которого будет создаваться новая база данных, либо `DEFAULT`, чтобы выбрать шаблон по умолчанию (`template1`).

`кодировка`

Кодировка символов в новой базе данных. Укажите строковую константу (например, `'SQL_ASCII'`) или целочисленный номер кодировки, либо `DEFAULT`, чтобы выбрать кодировку по умолчанию (а именно, кодировку шаблона). Наборы символов, которые поддерживает Postgres Pro, перечислены в Подразделе 22.3.1. Дополнительные ограничения описаны ниже.

`категория_сортировки`[@*`провайдер`*]

Порядок сортировки (`LC_COLLATE`), который будет использоваться в новой базе данных. Этот параметр определяет порядок сортировки строк, например, в запросах с ORDER BY, а также

порядок индексов по текстовым столбцам. По умолчанию используется порядок сортировки, установленный в шаблоне БД. Кроме того, вы можете задать провайдер основного правила сортировки после символа `@` (об этом рассказывается в Подразделе 22.2.2). Допустимые варианты этого указания: `icu` и `libc`. Дополнительные ограничения описаны ниже.

категория_типов_символов

Классификация символов (`LC_CTYPE`), которая будет применяться в новой базе данных. Этот параметр определяет принадлежность символов категориям, например: строчные, заглавные, цифры и т. п. По умолчанию используется классификация символов, установленная в шаблоне. Дополнительные ограничения описаны ниже.

табл_пространство

Имя табличного пространства, связываемого с новой базой данных, или `DEFAULT` для использования табличного пространства шаблона. Это табличное пространство будет использоваться по умолчанию для объектов, создаваемых в этой базе. За подробностями обратитесь к CREATE TABLESPACE.

разр_подключения

Если false, никто не сможет подключаться к этой базе данных. По умолчанию имеет значение true, то есть подключения принимаются (если не ограничиваются другими механизмами, например, `GRANT`/`REVOKE CONNECT`).

предел_подключений

Максимальное количество одновременных подключений к этой базе данных. Значение -1 (по умолчанию) снимает ограничение.

это_шаблон

Если true, базу данных сможет клонировать любой пользователь с правами `CREATEDB`; в противном случае (по умолчанию), клонировать эту базу смогут только суперпользователи и её владелец.

Дополнительные параметры могут записываться в любом порядке, не обязательно так, как показано выше.

Замечания

`CREATE DATABASE` нельзя выполнять внутри блока транзакции.

Ошибки, содержащие сообщение «не удалось инициализировать каталог базы данных», чаще всего связаны с нехваткой прав в каталоге данных, заполнением диска или другими проблемами в файловой системе.

Для удаления базы данных применяется DROP DATABASE.

Программа createdb представляет собой оболочку этой команды, созданную ради удобства.

Конфигурационные параметры уровня базы данных (устанавливаемые командой ALTER DATABASE) и разрешения уровня базы (устанавливаемые командой GRANT) из шаблона не копируются.

Хотя с помощью этой команды можно скопировать любую базу данных, а не только `template1`, указав её имя в качестве имени шаблона, она не предназначена (пока) для использования в качестве универсального средства вроде «`COPY DATABASE`». Принципиальным ограничением является невозможность копирования базы данных шаблона, если установлены другие подключения к ней. `CREATE DATABASE` выдаёт ошибку, если при запуске команды есть другие подключения к этой базе; в противном случае новые подключения к базе блокируются до завершения команды `CREATE DATABASE`. За дополнительными сведениями обратитесь к Разделу 21.3.

Кодировка символов, указанная для новой базы данных, должна быть совместима с выбранными параметрами локали (LC_COLLATE и LC_CTYPE). Если выбрана локаль C (или равнозначная ей POSIX), допускаются все кодировки, но для других локалей правильно будет работать только одна кодировка. (В Windows, однако, кодировку UTF-8 можно использовать с любой локалью.) CREATE DATABASE позволяет суперпользователям указать кодировку SQL_ASCII вне зависимости от локали, но этот вариант считается устаревшим и может привести к ошибочному поведению строковых функций, если в базе хранятся данные в кодировке, несовместимой с заданной локалью.

Параметры локали и кодировка должны соответствовать тем, что установлены в шаблоне, если только это не template0. Это ограничение объясняется тем, что другие базы данных могут содержать данные в кодировке, отличной от заданной, или индексы, порядок сортировки которых определяются параметрами LC_COLLATE и LC_CTYPE. При копировании таких данных получится база, которая будет испорченной согласно новым параметрам локали. Однако template0 определённо не содержит какие-либо данные или индексы, зависящие от кодировки или локали.

Ограничение CONNECTION LIMIT действует только приблизительно; если одновременно запускаются два сеанса, тогда как в базе остаётся только одно «свободное место», может так случиться, что будут отклонены оба подключения. Кроме того, это ограничение не распространяется на суперпользователей и фоновые рабочие процессы.

Примеры

Создание базы данных:

```
CREATE DATABASE lusiadas;
```

Создание базы данных sales, принадлежащей пользователю salesapp, с табличным пространством по умолчанию salesspace:

```
CREATE DATABASE sales OWNER salesapp TABLESPACE salesspace;
```

Создание базы данных music с другой локалью:

```
CREATE DATABASE music
    LC_COLLATE 'sv_SE.utf8' LC_CTYPE 'sv_SE.utf8'
    TEMPLATE template0;
```

В этом примере предложение TEMPLATE template0 необходимо, только если указанная локаль отличается от локали в template1. (В противном случае явное указание локали является избыточным.)

Создание базы данных music2 с другой локалью и другой кодировкой символов:

```
CREATE DATABASE music2
    LC_COLLATE 'sv_SE.iso885915' LC_CTYPE 'sv_SE.iso885915'
    ENCODING LATIN9
    TEMPLATE template0;
```

Свойства кодировки должны соответствовать локали, иначе возникнет ошибка.

Заметьте, что имена локалей зависят от операционной системы, так что показанные выше команды могут не везде работать одинаково.

Совместимость

Оператор CREATE DATABASE отсутствует в стандарте SQL. Базы данных равнозначны каталогам, а их создание в стандарте определяется реализацией.

См. также

ALTER DATABASE, DROP DATABASE

CREATE DOMAIN

CREATE DOMAIN — создать домен

Синтаксис

```
CREATE DOMAIN имя [ AS ] тип_данных
    [ COLLATE правило_сортировки ]
    [ DEFAULT выражение ]
    [ ограничение [ ... ] ]

Здесь ограничение:

[ CONSTRAINT имя_ограничения ]
{ NOT NULL | NULL | CHECK (выражение) }
```

Описание

`CREATE DOMAIN` создаёт новый домен. Домен по сути представляет собой тип данных с дополнительными условиями (ограничивающими допустимый набор значений). Владельцем домена становится пользователь его создавший.

Если задаётся имя схемы (например, `CREATE DOMAIN myschema.mydomain ...`), домен создаётся в указанной схеме, в противном случае — в текущей. Имя домена должно быть уникальным среди имён типов и доменов, существующих в этой схеме.

Домены полезны для абстрагирования и вынесения общих характеристик разных полей в единое место для упрощения сопровождения. Например, в нескольких таблицах может присутствовать столбец, содержащий электронный адрес, и для всех требуются одинаковые ограничения CHECK, проверяющие синтаксис адреса. В этом случае лучше определить домен, а не задавать для каждой таблицы отдельные ограничения.

Чтобы создать домен, необходимо иметь право `USAGE` для нижележащего типа.

Параметры

имя

Имя создаваемого домена (возможно, дополненное схемой).

тип_данных

Нижележащий тип данных домена (может включать определение массива с этим типом).

правило_сортировки

Необязательное указание правила сортировки для домена. Если это указание отсутствует, в домене используется правило сортировки нижележащего типа данных. Указать `COLLATE` можно, только если нижележащий тип данных является сортируемым.

`DEFAULT` *выражение*

Предложение `DEFAULT` определяет значение по умолчанию для столбцов, типом данных которых является этот домен. Значением может быть любое выражение без переменных (подзапросы также не допускаются). Тип данных этого выражения должен соответствовать типу данных домена. Если значение по умолчанию не указано, им будет значение NULL.

Значение по умолчанию будет использоваться в любой операции добавления строк, в которой не задано значение для этого столбца. Если значение по умолчанию установлено для конкретного столбца, оно будет переопределять значение по умолчанию, связанное с доменом.

В свою очередь, значение по умолчанию для домена переопределяет любое значение по умолчанию, связанное с нижележащим типом данных.

`CONSTRAINT` *`имя_ограничения`*

Имя ограничения. Если не указано явно, имя будет сгенерировано системой.

`NOT NULL`

Значения этого домена будут отличны от NULL (но см. замечания ниже).

`NULL`

Этот домен может содержать значение NULL. Это свойство домена по умолчанию.

Это предложение предназначено только для совместимости с нестандартными базами данных SQL. Использовать его в новых приложениях не рекомендуется.

`CHECK (`*`выражение`*`)`

Предложения `CHECK` задают ограничения целостности или проверки, которым должны удовлетворять значения домена. Каждое ограничение должно представлять собой выражение, выдающее результат типа Boolean. Проверяемое значение в этом выражении обозначается ключевым словом `VALUE`. Если выражение выдаёт FALSE, сообщается об ошибке и приведение значения к типу домена запрещается.

В настоящее время выражения `CHECK` не могут содержать переменные, кроме `VALUE`, и подзапросы.

Когда для домена задано несколько ограничений `CHECK`, они будут проверяться в алфавитном порядке имён. (До версии 9.5 в PostgreSQL не было установлено никакого определённого порядка обработки ограничений `CHECK`.)

Замечания

Ограничения домена, в частности `NOT NULL`, проверяются при преобразовании значения к типу домена. Однако из столбца, который номинально имеет тип домена, всё же можно прочитать NULL, несмотря на такое ограничение. Например, это может происходить в запросе внешнего соединения, если столбец домена окажется в обнуляемой стороне внешнего соединения. Более тонкий пример:

```
INSERT INTO tab (domcol) VALUES ((SELECT domcol FROM tab WHERE false));
```

Пустой скалярный вложенный SELECT выдаст значение NULL, типом которого будет считаться домен, так что к этому значению не будут применены дополнительные проверки ограничений и строка будет успешно добавлена.

Избежать таких проблем очень сложно, так как в SQL вообще предполагается, что значение NULL является подходящим для любого типа данных. Таким образом, лучше всего разрабатывать ограничения так, чтобы значения NULL допускались, а затем при необходимости применять ограничения `NOT NULL` к столбцам доменного типа, а не непосредственно к самому этому типу.

В Postgres Pro предполагается, что условия ограничений `CHECK` являются постоянными, то есть при одинаковых входных значениях они всегда выдают одинаковый результат. Именно этим предположением оправдывается то, что ограничения `CHECK` проверяются только при первом преобразовании значения в тип домена, а не при каждом обращении к нему. (По сути таким же образом обрабатываются ограничения `CHECK` для таблиц, как описано в Подразделе 5.3.1.)

Однако это предположение может нарушаться, как часто бывает, когда в выражении `CHECK` используется пользовательская функция, поведение которой впоследствии меняется. Postgres Pro не запрещает этого, и если сохранённые значения типа домена перестанут удовлетворять ограничению `CHECK`, это останется незамеченным. В итоге при попытке загрузить выгруженные позже данные могут возникнуть проблемы. Поэтому подобные изменения рекомендуется

осуществлять следующим образом: удалить ограничение (используя `ALTER DOMAIN`), изменить определение функции, а затем пересоздать ограничение той же командой, которая при этом перепроверит сохранённые данные.

Примеры

В этом примере создаётся тип данных `us_postal_code` (почтовый индекс США), который затем используется в определении таблицы. Для проверки значения на соответствие формату почтовых индексов США применяется проверка с регулярными выражениями:

```
CREATE DOMAIN us_postal_code AS TEXT
CHECK(
   VALUE ~ '^\d{5}$'
OR VALUE ~ '^\d{5}-\d{4}$'
);

CREATE TABLE us_snail_addy (
  address_id SERIAL PRIMARY KEY,
  street1 TEXT NOT NULL,
  street2 TEXT,
  street3 TEXT,
  city TEXT NOT NULL,
  postal us_postal_code NOT NULL
);
```

Совместимость

Команда `CREATE DOMAIN` соответствует стандарту SQL.

См. также

ALTER DOMAIN, DROP DOMAIN

CREATE EVENT TRIGGER

CREATE EVENT TRIGGER — создать событийный триггер

Синтаксис

```
CREATE EVENT TRIGGER имя
    ON событие
    [ WHEN переменная_фильтра IN (значение_фильтра [, ... ]) [ AND ... ] ]
    EXECUTE PROCEDURE имя_функции()
```

Описание

`CREATE EVENT TRIGGER` создаёт новый событийный триггер. Функция триггера выполняется, когда происходит указанное событие и удовлетворяется связанное с триггером условие `WHEN` (если такое имеется). За вводной информацией по триггерам обратитесь к Главе 37. Владельцем триггера становится пользователь его создавший.

Параметры

имя

Имя, назначаемое новому триггеру. Это имя должно быть уникальным в базе данных.

событие

Имя события, при котором срабатывает триггер и вызывается заданная функция. Подробнее об именах событий можно узнать в Разделе 37.1.

переменная_фильтра

Имя переменной, применяемой для фильтрования событий. Это указание позволяет ограничить срабатывание триггера подмножеством случаев, в которых он поддерживается. В настоящее время единственно возможное значение параметра *переменная_фильтра* — `TAG`.

значение_фильтра

Список значений связанного параметра *переменная_фильтра*, для которых должен срабатывать триггер. Для переменной `TAG` это список меток команд (например, `'DROP FUNCTION'`).

имя_функции

Заданная пользователем функция, объявленная как функция без аргументов и возвращающая тип `event_trigger`.

Замечания

Создавать событийные триггеры могут только суперпользователи.

Событийные триггеры не вызываются в однопользовательском режиме (см. postgres). Если ошибочный событийный триггер заблокировал работу с базой данных так, что даже удалить его нельзя, перезапустите сервер в однопользовательском режиме и это можно будет сделать.

Примеры

Триггер, запрещающий выполнение любой команды DDL:

```
CREATE OR REPLACE FUNCTION abort_any_command()
  RETURNS event_trigger
 LANGUAGE plpgsql
  AS $$
```

```
BEGIN
  RAISE EXCEPTION 'команда % отключена', tg_tag;
END;
$$;

CREATE EVENT TRIGGER abort_ddl ON ddl_command_start
   EXECUTE PROCEDURE abort_any_command();
```

Совместимость

Оператор `CREATE EVENT TRIGGER` отсутствует в стандарте SQL.

См. также

ALTER EVENT TRIGGER, DROP EVENT TRIGGER, CREATE FUNCTION

CREATE EXTENSION

CREATE EXTENSION — установить расширение

Синтаксис

```
CREATE EXTENSION [ IF NOT EXISTS ] имя_расширения
    [ WITH ] [ SCHEMA имя_схемы ]
             [ VERSION версия ]
             [ FROM старая_версия ]
             [ CASCADE ]
```

Описание

`CREATE EXTENSION` загружает в текущую базу данных новое расширение. Расширение с таким именем не должно быть уже загружено.

Загрузка расширения по сути сводится к запуску скрипта расширения. Этот скрипт обычно создаёт новые SQL-объекты, такие как функции, типы данных, операторы и методы поддержки индексов. `CREATE EXTENSION` дополнительно записывает идентификаторы всех добавляемых объектов, так что впоследствии их можно удалить, выполнив команду `DROP EXTENSION`.

Для загрузки расширения требуются те же права, что необходимы для создания составляющих его объектов. Для большинства расширений это означает, что необходимы права владельца базы данных или суперпользователя. Пользователь, запускающий `CREATE EXTENSION`, становится владельцем самого расширения (это требуется для последующих проверок доступа), а также владельцем всех объектов, созданных скриптом расширения.

Параметры

`IF NOT EXISTS`

Не считать ошибкой, если расширение с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее расширение как-то соотносится с тем, которое могло бы быть создано из указанного скрипта.

имя_расширения

Имя устанавливаемого расширения. Postgres Pro создаст расширение, используя инструкции из файла `SHAREDIR/extension/`*имя_расширения*`.control`.

имя_схемы

Имя схемы, в которую будут установлены объекты расширения (подразумевается, что расширение позволяет управлять размещением своих объектов). Указанная схема должна уже существовать. Если имя не указано и в управляющем файле расширения оно так же не задано, для создания объектов используется текущая схема.

Если в управляющем файле расширения задаётся параметр `schema`, заданную схему нельзя переопределить предложением `SCHEMA`. Обычно при указании предложения `SCHEMA` возникает ошибка, если эта схема конфликтует с параметром `schema` данного расширения. Однако если также задаётся предложение `CASCADE`, в случае конфликта *имя_схемы* игнорируется. Заданное *имя_схемы* будет использоваться для установки всех необходимых расширений, в управляющих файлах которых не задаётся `schema`.

Помните, что само расширение не считается принадлежащим какой-либо схеме; имена расширений не дополняются схемой и потому должны быть уникальными во всей базе данных. Однако объекты, принадлежащие расширениям, могут относиться к схемам.

версия

Версия устанавливаемого расширения. Её можно записать в виде идентификатора или строкового значения. По умолчанию версия считывается из управляющего файла расширения.

старая_версия

Указание FROM *старая_версия* может быть добавлено тогда и только тогда, когда устанавливаемое расширение заменяет модуль «старого стиля», представляющий собой просто набор объектов, не упакованный в расширение. С этим указанием CREATE EXTENSION запускает альтернативный установочный скрипт, собирающий все существующие объекты в расширение, а не создающий новые. Учтите, что SCHEMA при этом определяет схему, содержащую эти существующие объекты.

Значение, задаваемое в качестве *старой_версии*, определяется автором расширения и может меняться, если в расширение нужно преобразовать не одну версию модуля в старом стиле. Для стандартных дополнительных модулей, поставляемых в PostgreSQL до версии 9.1, при преобразовании модуля в расширение *старая_версия* должна содержать значение unpackaged.

CASCADE

Автоматически устанавливать все расширения, от которого зависит данное, если они ещё не установлены. Их зависимости подобным образом рекурсивно устанавливаются автоматически. Предложение SCHEMA, если задано, применяется ко всем расширениям, устанавливаемым таким способом. Другие параметры оператора к автоматически устанавливаемым расширениям не применяются; в частности, всегда выбираются их версии по умолчанию.

Замечания

Прежде чем вы сможете выполнить CREATE EXTENSION и загрузить расширение в базу данных, необходимо правильно установить сопутствующие файлы расширения. Информацию об установке расширений, поставляемых в составе Postgres Pro, можно найти по ссылке Дополнительные поставляемые модули.

Расширения, доступные для установки в данный момент, можно найти в системном представлении pg_available_extensions или pg_available_extension_versions.

Внимание

Устанавливая расширение от имени суперпользователя, важно иметь уверенность в том, что автор расширения написал установочный скрипт безопасным образом. Для злонамеренного пользователя не составит большого труда создать объект типа троянского коня, который впоследствии скомпрометирует выполнение неаккуратно написанного скрипта расширения и позволит этому пользователю стать суперпользователем. Однако такие объекты опасны, только если они находятся в пути search_path во время выполнения скрипта, то есть, если они находятся в схеме, в которую устанавливается расширение, или в схеме, где располагается другое расширение, от которого зависит первое. Таким образом, имея дело с расширениями, скрипты которых не были тщательно проверены, рекомендуется устанавливать их только в те схемы, в которых недоверенные пользователи не имеют и не будут иметь права CREATE. То же самое касается их зависимостей, других расширений.

Расширения, поставляемые в составе Postgres Pro, можно считать защищёнными от подобного рода атак времени выполнения, за исключением некоторых, зависящих от других расширений. Как отмечается в документации таких расширений, их следует устанавливать в безопасные схемы и/или в те же схемы, в которые устанавливаются требующиеся им расширения.

За информацией для разработчиков расширений обратитесь к Разделу 35.15.

Примеры

Установка расширения hstore в текущую базу данных (при этом объекты расширения размещаются в схеме `addons`):

```
CREATE EXTENSION hstore SCHEMA addons;
```

Другой способ сделать то же самое:

```
SET search_path = addons;
CREATE EXTENSION hstore;
```

Преобразование установленного до версии 9.1 модуля `hstore` в расширение:

```
CREATE EXTENSION hstore SCHEMA public FROM unpackaged;
```

Будьте внимательны — здесь нужно указать схему, в которую ранее были установлены существующие объекты `hstore`.

Совместимость

`CREATE EXTENSION` является расширением Postgres Pro.

См. также

ALTER EXTENSION, DROP EXTENSION

CREATE FOREIGN DATA WRAPPER

CREATE FOREIGN DATA WRAPPER — создать новую обёртку сторонних данных

Синтаксис

```
CREATE FOREIGN DATA WRAPPER имя
    [ HANDLER функция_обработчик | NO HANDLER ]
    [ VALIDATOR функция_проверки | NO VALIDATOR ]
    [ OPTIONS ( параметр 'значение' [, ... ] ) ]
```

Описание

`CREATE FOREIGN DATA WRAPPER` создаёт обёртку сторонних данных. Владельцем обёртки становится создавший её пользователь.

Имя обёртки сторонних данных должно быть уникальным в базе данных.

Создавать обёртки сторонних данных могут только суперпользователи.

Параметры

имя

Имя создаваемой обёртки сторонних данных.

`HANDLER` *функция_обработчик*

В аргументе *функция_обработчик* указывается имя ранее зарегистрированной функции, которая будет вызываться для получения функций, реализующих обращения к сторонним таблицам. Функция-обработчик не принимает аргументы и возвращает результат типа `fdw_handler`.

Обёртку сторонних таблиц можно создать и без функции-обработчика, но через такую обёртку нельзя будет использовать сторонние таблицы, хотя объявить их вполне возможно.

`VALIDATOR` *функция_проверки*

В аргументе *функция_проверки* указывается имя ранее зарегистрированной функции, которая будет вызываться для проверки общих параметров, передаваемых обёртке сторонних данных, а также параметров сторонних серверов, сопоставлений пользователей и сторонних таблиц, доступных через эту обёртку. Если функция проверки не задана или указано `NO VALIDATOR`, параметры не будут проверяться во время создания объектов. (Обёртка сторонних данных может игнорировать или не принимать неверные указания параметров во время выполнения, в зависимости от реализации.) Функция проверки должна принимать два аргумента: первый типа `text[]` (в нём содержится массив параметров, хранящихся в системном каталоге), а второй типа `oid` (в нём указывается OID системного каталога с этими параметрами). Возвращаемое значение игнорируется; функция проверки должна сообщать о неверных параметрах, вызывая системную функцию `ereport(ERROR)`.

`OPTIONS ( `*параметр* `'`*значение*`' [, ... ] )`

Это предложение определяет параметры для создаваемой обёртки сторонних данных. Набор допустимых параметров и значений для каждой обёртки свой, контроль их правильности осуществляет функция проверки сторонних данных. Имена параметров должны быть уникальными.

Замечания

Функциональность Postgres Pro по работе со сторонними данными продолжает активно развиваться. На данный момент выполняется только примитивная оптимизация запросов (и по

большей части это тоже делает обёртка), так что в этом направлении есть поле для улучшения производительности.

Примеры

Создание бесполезной обёртки сторонних данных `dummy`:

```
CREATE FOREIGN DATA WRAPPER dummy;
```

Создание обёртки сторонних данных `file` с функцией-обработчиком `file_fdw_handler`:

```
CREATE FOREIGN DATA WRAPPER file HANDLER file_fdw_handler;
```

Создание обёртки сторонних данных `mywrapper` с параметрами:

```
CREATE FOREIGN DATA WRAPPER mywrapper
    OPTIONS (debug 'true');
```

Совместимость

`CREATE FOREIGN DATA WRAPPER` соответствует стандарту ISO/IEC 9075-9 (SQL/MED), за исключением того, что предложения `HANDLER` и `VALIDATOR` стандартом не предусмотрены, а предложения `LIBRARY` и `LANGUAGE`, напротив, не реализованы в Postgres Pro.

Учтите, однако, что функциональность SQL/MED в целом ещё не обеспечивается.

См. также

ALTER FOREIGN DATA WRAPPER, DROP FOREIGN DATA WRAPPER, CREATE SERVER, CREATE USER MAPPING, CREATE FOREIGN TABLE

CREATE FOREIGN TABLE

CREATE FOREIGN TABLE — создать стороннюю таблицу

Синтаксис

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] имя_таблицы ( [
  { имя_столбца тип_данных [ OPTIONS ( параметр 'значение' [, ... ] ) ]
 [ COLLATE правило_сортировки ] [ ограничение_столбца [ ... ] ]
    | ограничение_таблицы }
    [, ... ]
] )
[ INHERITS ( таблица_родитель [, ... ] ) ]
  SERVER имя_сервера
[ OPTIONS ( параметр 'значение' [, ... ] ) ]

CREATE FOREIGN TABLE [ IF NOT EXISTS ] имя_таблицы
  PARTITION OF таблица_родитель [ (
  { имя_столбца [ WITH OPTIONS ] [ ограничение_столбца [ ... ] ]
    | ограничение_таблицы }
    [, ... ]
) ]
  FOR VALUES указание_границ_секции
  SERVER имя_сервера
[ OPTIONS ( параметр 'значение' [, ... ] ) ]

Здесь ограничение_столбца:

[ CONSTRAINT имя_ограничения ]
{ NOT NULL |
  NULL |
  CHECK ( выражение ) [ NO INHERIT ] |
  DEFAULT выражение_по_умолчанию }

и ограничение_таблицы:

[ CONSTRAINT имя_ограничения ]
CHECK ( выражение ) [ NO INHERIT ]

и указание_границ_секции:

IN ( { числовая_константа | строковая_константа | TRUE | FALSE | NULL } [, ...] ) |
FROM ( { числовая_константа | строковая_константа | TRUE | FALSE | MINVALUE |
 MAXVALUE } [, ...] )
  TO ( { числовая_константа | строковая_константа | TRUE | FALSE | MINVALUE |
 MAXVALUE } [, ...] )
```

Описание

`CREATE FOREIGN TABLE` создаёт новую стороннюю таблицу в текущей базе данных. Владельцем таблицы будет пользователь, выполнивший эту команду.

Если указано имя схемы (например, `CREATE FOREIGN TABLE myschema.mytable ...`), таблица будет создана в этой схеме. В противном случае она создаётся в текущей схеме. Имя сторонней таблицы должно отличаться от имён других сторонних и обычных таблиц, последовательностей, индексов, представлений и материализованных представлений, существующих в этой схеме.

`CREATE FOREIGN TABLE` также автоматически создаёт составной тип данных, соответствующий одной строке сторонней таблицы. Таким образом, имя сторонней таблицы не может совпадать с именем существующего типа в этой же схеме.

Если указано предложение `PARTITION OF`, таблица создаётся в виде секции `parent_table` с указанными границами.

Чтобы создать стороннюю таблицу, необходимо иметь право `USAGE` для стороннего сервера, а также право `USAGE` для всех типов столбцов, содержащихся в таблице.

Параметры

`IF NOT EXISTS`

Не считать ошибкой, если отношение с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее отношение как-то соотносится с тем, которое могло бы быть создано.

`имя_таблицы`

Имя создаваемой таблицы (возможно, дополненное схемой).

`имя_столбца`

Имя столбца, создаваемого в новой таблице.

`тип_данных`

Тип данных столбца (может включать определение массива с этим типом). За дополнительными сведениями о типах данных, которые поддерживает Postgres Pro, обратитесь к Главе 8.

`COLLATE` *`правило_сортировки`*

Предложение `COLLATE` назначает правило сортировки для столбца (который должен иметь тип, поддерживающий сортировку). Если оно отсутствует, используется правило сортировки по умолчанию, установленное для типа данных столбца.

`INHERITS ( таблица_родитель [, ... ] )`

Необязательное предложение `INHERITS` определяет список таблиц, от которых новая сторонняя таблица будет автоматически наследовать все столбцы. Родительскими таблицами могут быть обычные или сторонние таблицы. За подробностями обратитесь к описанию подобной формы CREATE TABLE.

`PARTITION OF` *`таблица_родитель`* `FOR VALUES` *`указание_границ_секции`*

Эта форма может использоваться для создания сторонней таблицы в виде секции указанной родительской таблицы с заданными граничными значениями. За подробностями обратитесь к описанию подобной формы команды CREATE TABLE.

`CONSTRAINT` *`имя_ограничения`*

Необязательное имя столбца или ограничения таблицы. При нарушении ограничения его имя будет выводиться в сообщении об ошибках, так что имена ограничений вида `столбец должен быть положительным` могут сообщить полезную информацию об ограничении клиентскому приложению. (Имена ограничений, включающие пробелы, необходимо заключать в двойные кавычки.) Если имя ограничения не указано, система генерирует имя автоматически.

`NOT NULL`

Данный столбец не принимает значения NULL.

`NULL`

Данный столбец может содержать значения NULL (по умолчанию).

Это предложение предназначено только для совместимости с нестандартными базами данных SQL. Использовать его в новых приложениях не рекомендуется.

`CHECK ( выражение ) [ NO INHERIT ]`

В ограничении `CHECK` задаётся выражение, возвращающее логический результат, которому должны удовлетворять все строки в сторонней таблице; то есть это выражение должно выдавать TRUE или UNKNOWN, но никогда FALSE, для всех строк в сторонней таблице. Ограничение-проверка, заданное как ограничение столбца, должно ссылаться только на значение самого столбца, тогда как ограничение на уровне таблицы может ссылаться и на несколько столбцов.

В настоящее время выражения `CHECK` не могут содержать подзапросы или ссылаться на переменные, кроме как на столбцы текущей строки. Также допустима ссылка на системный столбец `tableoid`, но не на другие системные столбцы.

Ограничение с пометкой `NO INHERIT` не будет наследоваться дочерними таблицами.

`DEFAULT выражение_по_умолчанию`

Предложение `DEFAULT` задаёт значение по умолчанию для столбца, в определении которого оно присутствует. Значение задаётся выражением без переменных (подзапросы и перекрёстные ссылки на другие столбцы текущей таблицы в нём не допускаются). Тип данных выражения, задающего значение по умолчанию, должен соответствовать типу данных столбца.

Это выражение будет использоваться во всех операциях добавления данных, в которых не задаётся значение данного столбца. Если значение по умолчанию не определено, таким значением будет NULL.

имя_сервера

Имя существующего стороннего сервера, предоставляющего данную стороннюю таблицу. О создании сервера можно узнать в CREATE SERVER.

`OPTIONS ( параметр 'значение' [, ...] )`

Параметры, связываемые с новой сторонней таблицей или одним из её столбцов. Допустимые имена и значения параметров у каждой обёртки сторонних данных свои; они контролируются функцией проверки, связанной с этой обёрткой. Имена параметров не должны повторяться (хотя параметр таблицы и параметр столбца вполне могут иметь одно имя).

Замечания

Ограничения сторонних таблиц (например, `CHECK` и `NOT NULL`) не контролируются ядром системы Postgres Pro, как не пытаются их контролировать и большинство обёрток сторонних данных; то есть, система просто предполагает, что ограничение выполняется. Контролировать такое ограничение не имело бы большого смысла, так как оно применялось бы только к строкам, добавляемым или изменяемым через стороннюю таблицу, но не к строкам, модифицируемым другим путём, например, непосредственно на удалённом сервере. Вместо этого, ограничение, связанное со сторонней таблицей, должно представлять ограничение, выполнение которого обеспечивает удалённый сервер.

Некоторые специализированные обёртки сторонних данных могут быть единственным вариантом обращения к доступным через них данным, и в этом случае может быть уместно реализовать контроль ограничений в самой такой обёртке. Но не следует полагать, что какая-либо обёртка ведёт себя так, если об этом не сказано явно в её документации.

Хотя Postgres Pro не пытается контролировать ограничения для сторонних таблиц, он полагает, что они выполняются для целей оптимизации запросов. Если в сторонней таблице будут видны строки, не удовлетворяющие объявленному ограничению, запросы к этой таблицы могут выдавать некорректные результаты. Ответственность за фактическое выполнение условия ограничения лежит на пользователе.

Хотя сторонние таблицы могут создаваться как секции, перенаправление кортежей в секции, представленные сторонними таблицами, не поддерживается.

Примеры

Создание сторонней таблицы `films`, которая будет доступна через сервер `film_server`:

```
CREATE FOREIGN TABLE films (
    code        char(5) NOT NULL,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
)
SERVER film_server;
```

Создание сторонней таблицы `measurement_y2016m07`, которая будет доступна через сервер `server_07`, в виде секции таблицы `measurement`, секционированной по диапазонам:

```
CREATE FOREIGN TABLE measurement_y2016m07
    PARTITION OF measurement FOR VALUES FROM ('2016-07-01') TO ('2016-08-01')
    SERVER server_07;
```

Совместимость

Команда `CREATE FOREIGN TABLE` в основном соответствует стандарту SQL; однако, как и `CREATE TABLE`, она допускает ограничения `NULL` и сторонние таблицы с нулём столбцов. Возможность задавать значения по умолчанию для столбцов также является расширением Postgres Pro. Наследование таблиц, в форме, определённой в Postgres Pro, стандарту не соответствует.

См. также

ALTER FOREIGN TABLE, DROP FOREIGN TABLE, CREATE TABLE, CREATE SERVER, IMPORT FOREIGN SCHEMA

CREATE FUNCTION

CREATE FUNCTION — создать функцию

Синтаксис

```
CREATE [ OR REPLACE ] FUNCTION
    имя ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ { DEFAULT |
 = } выражение_по_умолчанию ] [, ...] ] )
    [ RETURNS тип_результата
      | RETURNS TABLE ( имя_столбца тип_столбца [, ...] ) ]
  { LANGUAGE имя_языка
    | TRANSFORM { FOR TYPE имя_типа } [, ... ]
    | WINDOW
    | { IMMUTABLE | STABLE | VOLATILE }
    | [ NOT ] LEAKPROOF
    | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
    | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }
    | PARALLEL { UNSAFE | RESTRICTED | SAFE }
    | COST стоимость_выполнения
    | ROWS строк_в_результате
    | SET параметр_конфигурации { TO значение | = значение | FROM CURRENT }
    | AS 'определение'
    | AS 'объектный_файл', 'объектный_символ'
  } ...
    [ WITH ( атрибут [, ...] ) ]
```

Описание

Команда `CREATE FUNCTION` определяет новую функцию. `CREATE OR REPLACE FUNCTION` создаёт новую функцию, либо заменяет определение уже существующей. Чтобы определить функцию, необходимо иметь право `USAGE` для соответствующего языка.

Если указано имя схемы, функция создаётся в заданной схеме, в противном случае — в текущей. Имя новой функции должно отличаться от имён существующих функций с такими же типами аргументов в этой схеме. Однако функции с аргументами разных типов могут иметь одно имя (это называется *перегрузкой*).

Чтобы заменить текущее определение существующей функции, используйте команду `CREATE OR REPLACE FUNCTION`. Но учтите, что она не позволяет изменить имя или аргументы функции (если попытаться сделать это, на самом деле будет создана новая, независимая функция). Кроме того, `CREATE OR REPLACE FUNCTION` не позволит изменить тип результата существующей функции. Чтобы сделать это, придётся удалить функцию и создать её заново. (Это означает, что если функция имеет выходные параметры (`OUT`), то изменить типы параметров `OUT` можно, только удалив функцию.)

Когда команда `CREATE OR REPLACE FUNCTION` заменяет существующую функцию, владелец и права доступа к этой функции не меняются. Все другие свойства функции получают значения, задаваемые командой явно или по умолчанию. Чтобы заменить функцию, необходимо быть её владельцем (или быть членом роли-владельца).

Если вы удалите и затем вновь создадите функцию, новая функция станет другой сущностью, отличной от старой; вам потребуется так же удалить существующие правила, представления, триггеры и т. п., ссылающиеся на старую функцию. Поэтому, чтобы изменить определение функции, сохраняя ссылающиеся на неё объекты, следует использовать `CREATE OR REPLACE FUNCTION`. Кроме того, многие дополнительные свойства существующей функции можно изменить с помощью `ALTER FUNCTION`.

Владельцем функции становится создавший её пользователь.

Чтобы создать функцию, необходимо иметь право USAGE для типов её аргументов и возвращаемого типа.

Параметры

имя

Имя создаваемой функции (возможно, дополненное схемой).

режим_аргумента

Режим аргумента: IN (входной), OUT (выходной), INOUT (входной и выходной) или VARIADIC (переменный). По умолчанию подразумевается IN. За единственным аргументом VARIADIC могут следовать только аргументы OUT. Кроме того, аргументы OUT и INOUT нельзя использовать с предложением RETURNS TABLE.

имя_аргумента

Имя аргумента. Некоторые языки (включая SQL и PL/pgSQL) позволяют использовать это имя в теле функции. Для других языков это имя служит просто дополнительным описанием, если говорить о самой функции; однако вы можете указывать имена аргументов при вызове функции для улучшения читаемости (см. Раздел 4.3). Имя выходного аргумента в любом случае имеет значение, так как оно определяет имя столбца в типе результата. (Если вы опустите имя выходного аргумента, система выберет для него имя по умолчанию.)

тип_аргумента

Тип данных аргумента функции (возможно, дополненный схемой), при наличии аргументов. Тип аргументов может быть базовым, составным или доменным, либо это может быть ссылка на столбец таблицы.

В зависимости от языка реализации также может допускаться указание «псевдотипов», например, cstring. Псевдотипы показывают, что фактический тип аргумента либо определён не полностью, либо существует вне множества обычных типов SQL.

Ссылка на тип столбца записывается в виде *имя_таблицы*.*имя_столбца*%TYPE. Иногда такое указание бывает полезно, так как позволяет создать функцию, независящую от изменений в определении таблицы.

выражение_по_умолчанию

Выражение, используемое для вычисления значения по умолчанию, если параметр не задан явно. Результат выражения должен сводиться к типу соответствующего параметра. Значения по умолчанию могут иметь только входные параметры (включая INOUT). Для всех входных параметров, следующих за параметром с определённым значением по умолчанию, также должны быть определены значения по умолчанию.

тип_результата

Тип возвращаемых данных (возможно, дополненный схемой). Это может быть базовый, составной или доменный тип, либо ссылка на тип столбца таблицы. В зависимости от языка реализации здесь также могут допускаться «псевдотипы», например cstring. Если функция не должна возвращать значение, в качестве типа результата указывается void.

В случае наличия параметров OUT или INOUT, предложение RETURNS можно опустить. Если оно присутствует, оно должно согласовываться с типом результата, выводимым из выходных параметров: в качестве возвращаемого типа указывается RECORD, если выходных параметров несколько, либо тип единственного выходного параметра.

Указание SETOF показывает, что функция возвращает множество, а не единственный элемент.

Ссылка на тип столбца записывается в виде *имя_таблицы*.*имя_столбца*%TYPE.

имя_столбца

Имя выходного столбца в записи RETURNS TABLE. По сути это ещё один способ объявить именованный выходной параметр (OUT), но RETURNS TABLE также подразумевает и RETURNS SETOF.

тип_столбца

Тип данных выходного столбца в записи RETURNS TABLE.

имя_языка

Имя языка, на котором реализована функция. Это может быть sql, c, internal, либо имя процедурного языка, определённого пользователем, например, plpgsql. Стиль написания этого имени в апострофах считается устаревшим и требует точного совпадения регистра.

TRANSFORM { FOR TYPE *имя_типа* } [, ...] }

Устанавливает список трансформаций, которые должны применяться при вызове функции. Трансформации выполняют преобразования между типами SQL и типами данных, специфичными для языков; см. CREATE TRANSFORM. Преобразования встроенных типов обычно жёстко предопределены в реализациях процедурных языков, так что их здесь указывать не нужно. Если реализация процедурного языка не может обработать тип и трансформация для него отсутствует, будет выполнено преобразование типов по умолчанию, но это зависит от реализации.

WINDOW

Указание WINDOW показывает, что создаётся не простая, а *оконная функция*. В настоящее время это имеет смысл только для функций, написанных на C. Атрибут WINDOW нельзя изменить, модифицируя впоследствии определение функции.

IMMUTABLE
STABLE
VOLATILE

Эти атрибуты информируют оптимизатор запросов о поведении функции. Одновременно можно указать не более одного атрибута. Если никакой атрибут не задан, по умолчанию подразумевается VOLATILE.

Характеристика IMMUTABLE (постоянная) показывает, что функция не может модифицировать базу данных и всегда возвращает один и тот же результат при определённых значениях аргументов; то есть, она не обращается к базе данных и не использует информацию, не переданную ей явно в списке аргументов. Если функция имеет такую характеристику, любой её вызов с аргументами-константами можно немедленно заменить значением функции.

Характеристика STABLE (стабильная) показывает, что функция не может модифицировать базу данных и в рамках одного сканирования таблицы она всегда возвращает один и тот же результат для определённых значений аргументов, но этот результат может быть разным в разных операторах SQL. Это подходящий выбор для функций, результаты которых зависят от содержимого базы данных и настраиваемых параметров (например, текущего часового пояса). (Но этот вариант не подходит для триггеров AFTER, желающих прочитать строки, изменённые текущей командой.) Также заметьте, что функции семейства current_timestamp также считаются стабильными, так как их результаты не меняются внутри транзакции.

Характеристика VOLATILE (изменчивая) показывает, что результат функции может меняться даже в рамках одного сканирования таблицы, так что её вызовы нельзя оптимизировать. Изменчивы в этом смысле относительно немногие функции баз данных, например: random(), currval() и timeofday(). Но заметьте, что любая функция с побочными эффектами должна быть классифицирована как изменчивая, даже если её результат вполне предсказуем, чтобы её вызовы не были соптимизированы; пример такой функции: setval().

За дополнительными подробностями обратитесь к Разделу 35.6.

`LEAKPROOF`

Характеристика `LEAKPROOF` (герметичная) показывает, что функция не имеет побочных эффектов. Она не раскрывает информацию о своих аргументах, кроме как возвращая результат. Например, функция, которая выдаёт сообщение об ошибке с некоторыми, но не всеми значениями аргументов, либо выводит значения аргументов в сообщении об ошибке, не является герметичной. Это влияет на то, как система выполняет запросы к представлениям, созданным с барьером безопасности (с указанием `security_barrier`), или к таблицам с включённой защитой строк. Во избежание неконтролируемой утечки данных система будет проверять условия из политик защиты и определений представлений с барьерами безопасности перед любыми условиями, которые задаёт пользователь в самом запросе и в которых задействуются негерметичные функции. Функции и операторы, помеченные как герметичные, считаются доверенными и могут выполняться перед условиями из политик защиты и представлений с барьерами безопасности. При этом функции, которые не имеют аргументов или которым не передаются никакие аргументы из представления с барьером безопасности или таблицы, не требуется помечать как герметичные, чтобы они выполнялись до условий, связанных с безопасностью. См. CREATE VIEW и Раздел 38.5. Это свойство может установить только суперпользователь.

`CALLED ON NULL INPUT`
`RETURNS NULL ON NULL INPUT`
`STRICT`

`CALLED ON NULL INPUT` (по умолчанию) показывает, что функция будет вызвана как обычно, если среди её аргументов оказываются значения NULL. В этом случае ответственность за проверку значений NULL и соответствующую их обработку ложится на разработчика функции.

Указание `RETURNS NULL ON NULL INPUT` или `STRICT` показывает, что функция всегда возвращает NULL, получив NULL в одном из аргументов. Такая функция не будет вызываться с аргументами NULL, вместо этого автоматически будет полагаться результат NULL.

`[EXTERNAL] SECURITY INVOKER`
`[EXTERNAL] SECURITY DEFINER`

Характеристика `SECURITY INVOKER` (безопасность вызывающего) показывает, что функция будет выполняться с правами пользователя, вызвавшего её. Этот вариант подразумевается по умолчанию. Вариант `SECURITY DEFINER` (безопасность определившего) определяет, что функция выполняется с правами пользователя, владеющего ей.

Ключевое слово `EXTERNAL` (внешняя) допускается для соответствия стандарту SQL, но является необязательным, так как, в отличие от SQL, эта характеристика распространяется на все функции, а не только внешние.

`PARALLEL`

Указание `PARALLEL UNSAFE` означает, что эту функцию нельзя выполнять в параллельном режиме и присутствие такой функции в операторе SQL приводит к выбору последовательного плана выполнения. Это характеристика функции по умолчанию. Указание `PARALLEL RESTRICTED` означает, что функцию можно выполнять в параллельном режиме, но только в ведущем процессе группы. `PARALLEL SAFE` показывает, что функция безопасна для выполнения в параллельном режиме без ограничений.

Функции должны помечаться как небезопасные для параллельного выполнения, если они изменяют состояние базы данных, вносят изменения в транзакции, например, используя подтранзакции, обращаются к последовательностям или пытаются сохранять параметры (например, используя `setval`). Ограниченно параллельными должны помечаться функции, которые обращаются к временным таблицам, состоянию клиентского подключения, курсорам, подготовленным операторам или разнообразному состоянию обслуживающего процесса, которое система не может синхронизировать в параллельном режиме (например, `setseed`

может выполнять только ведущий процесс группы, так как изменения, внесённые другим процессом, не передаются ведущему). Вообще, если функция помечена как безопасная, тогда как она является ограниченной или небезопасной, либо если она помечена как ограниченно безопасная, не являясь безопасной, при попытке вызвать её в параллельном запросе она может выдавать ошибки или неверные результаты. Функции на языке C при неправильной пометке теоретически могут проявлять полностью неопределённое поведение, так как система никак не может защититься от произвольного кода на C, но чаще все они будут вести себя не хуже, чем любая другая функция. В случае сомнений функцию следует помечать как небезопасную (`UNSAFE`), что и имеет место по умолчанию.

стоимость_выполнения

Положительное число, задающее примерную стоимость выполнения функции, в единицах cpu_operator_cost. Если функция возвращает множество, это число задаёт стоимость для одной строки. Если стоимость не указана, для функций на C и внутренних функций она считается равной 1 единице, а для функций на всех других языках — 100 единицам. При больших значениях планировщик будет стараться не вызывать эту функцию чаще, чем это необходимо.

строк_в_результате

Положительное число, задающее примерное число строк, которое будет ожидать планировщик на выходе этой функции. Это указание допустимо, только если функция объявлена как возвращающая множество. Предполагаемое по умолчанию значение — 1000 строк.

параметр_конфигурации
значение

Предложение `SET` определяет, что при вызове функции указанный параметр конфигурации должен принять заданное значение, а затем восстановить своё предыдущее значение при завершении функции. Предложение `SET FROM CURRENT` сохраняет в качестве значения, которое будет применено при входе в функцию, значение, действующее в момент выполнения `CREATE FUNCTION`.

Если в определение функции добавлено `SET`, то действие команды `SET LOCAL`, выполняемой внутри функции для того же параметра, ограничивается телом функции: предыдущее значение параметра так же будет восстановлено при завершении функции. Однако обычная команда `SET` (без `LOCAL`) переопределяет предложение `SET`, как и предыдущую команду `SET LOCAL`: действие такой команды будет сохранено и после завершения функции, если только не произойдёт откат транзакции.

За подробными сведениями об именах и значениях параметров обратитесь к SET и Главе 18.

определение

Строковая константа, определяющая реализацию функции; её значение зависит от языка. Это может быть имя внутренней функции, путь к объектному файлу, команда SQL или код функции на процедурном языке.

Часто бывает полезно заключать определение функции в доллары (см. Подраздел 4.1.2.4), а не в традиционные апострофы. Если не использовать доллары, все апострофы и обратные косые черты в определении функции придётся экранировать, дублируя их.

объектный_файл, *объектный_символ*

Эта форма предложения `AS` применяется для динамически загружаемых функций на языке C, когда имя функции в коде C не совпадает с именем функции в SQL. Строка *объектный_файл* задаёт имя файла разделяемой библиотеки, содержащей скомпилированную функцию на C, и воспринимается как параметр команды LOAD. Строка *объектный_символ* задаёт символ скомпонованной функции, то есть имя функции в исходном коде на языке C. Если объектный символ опущен, предполагается, что он совпадает с именем определяемой SQL-функции. В C имена всех функций должны быть различными, поэтому перегружаемым функциям,

реализованным на C, нужно давать разные имена (например, включать в имена C обозначения типов аргументов).

Если повторные вызовы `CREATE FUNCTION` ссылаются на один и тот же объектный файл, он загружается в рамках сеанса только один раз. Чтобы выгрузить и загрузить этот файл снова (например, в процессе разработки), начните новый сеанс.

атрибут

Исторически сложившийся способ передавать дополнительную информацию о функции. В этой записи могут присутствовать следующие атрибуты:

`isStrict`

Равнозначно указанию `STRICT` или `RETURNS NULL ON NULL INPUT`.

`isCachable`

Свойство `isCachable` — устаревший эквивалент `IMMUTABLE`; оно всё ещё поддерживается ради обратной совместимости.

Имена атрибутов являются регистронезависимыми.

За дополнительной информацией о разработке функций обратитесь к Разделу 35.3.

Перегрузка

Postgres Pro допускает *перегрузку* функций; то есть, позволяет использовать одно имя для нескольких различных функций, если у них различаются типы входных аргументов. Независимо от того, используете вы эту возможность или нет, она требует предосторожности при вызове функций в базах данных, где одни пользователи не доверяют другим; см. Раздел 10.3.

Две функции считаются совпадающими, если они имеют одинаковые имена и типы *входных* аргументов, параметры `OUT` игнорируются. Таким образом, например, эти объявления вызовут конфликт:

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, out text) ...
```

Функции, имеющие разные типы аргументов, не будут считаться конфликтующими в момент создания, но предоставленные для них значения по умолчанию могут вызвать конфликт в момент использования. Например, рассмотрите следующие определения:

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, int default 42) ...
```

Вызов `foo(10)` завершится ошибкой из-за неоднозначности в выборе вызываемой функции.

Замечания

В объявлении аргументов функции и возвращаемого значения допускается полный синтаксис описания типа SQL. Однако модификаторы типа в скобках (например, поле точности для типа `numeric`) команда `CREATE FUNCTION` не учитывает. Так что, например, `CREATE FUNCTION foo (varchar(10)) ...` создаст такую же функцию, что и `CREATE FUNCTION foo (varchar) ...`.

При замене существующей функции с помощью `CREATE OR REPLACE FUNCTION` есть ограничения на изменения имён параметров. В частности, нельзя изменить имя, уже назначенное любому входному параметру (хотя можно добавить имена ранее безымянным параметрам). Также, если у функции более одного выходного параметра, нельзя изменять имена выходных параметров, так как это приведёт к изменению имён столбцов анонимного составного типа, описывающего результат функции. Эти ограничения позволяют гарантировать, что существующие вызовы функции не перестанут работать после её замены.

Если функция объявлена как `STRICT` с аргументом `VARIADIC`, при оценивании строгости проверяется, что весь переменный массив *в целом* не NULL. Если же в этом массиве содержатся элементы NULL, функция будет вызываться.

Примеры

Ниже приведено несколько простых вводных примеров. За дополнительными сведениями и примерами обратитесь к Разделу 35.3.

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

Функция увеличения целого числа на 1, использующая именованный аргумент, на языке PL/pgSQL:

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$
        BEGIN
                RETURN i + 1;
        END;
$$ LANGUAGE plpgsql;
```

Функция, возвращающая запись с несколькими выходными параметрами:

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

То же самое можно сделать более развёрнуто, явно объявив составной тип:

```
CREATE TYPE dup_result AS (f1 int, f2 text);

CREATE FUNCTION dup(int) RETURNS dup_result
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

Ещё один способ вернуть несколько столбцов — применить функцию `TABLE`:

```
CREATE FUNCTION dup(int) RETURNS TABLE(f1 int, f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

Однако пример с `TABLE` отличается от предыдущих, так как в нём функция на самом деле возвращает не одну, а *набор* записей.

Разработка защищённых функций `SECURITY DEFINER`

Так как функция `SECURITY DEFINER` выполняется с правами пользователя, владеющего ей, необходимо позаботиться о том, чтобы её нельзя было использовать не по назначению. В целях безопасности в пути search_path следует исключить любые схемы, доступные на запись недоверенным пользователям. Это не позволит злонамеренным пользователям создать свои объекты (например, таблицы, функции и операторы), которые замаскируют объекты, используемые функцией. Особенно важно в этом отношении исключить схему временных таблиц, которая по умолчанию просматривается первой, а право записи в неё по умолчанию имеют все. Соответствующую защиту можно организовать, поместив временную схему в конец списка поиска.

Для этого следует сделать pg_tempпоследней записью в search_path. Безопасное использование демонстрирует следующая функция:

```
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
RETURNS BOOLEAN AS $$
DECLARE passed BOOLEAN;
BEGIN
        SELECT  (pwd = $2) INTO passed
        FROM    pwds
        WHERE   username = $1;

        RETURN passed;
END;
$$  LANGUAGE plpgsql
    SECURITY DEFINER
    -- Установить безопасный путь поиска: сначала доверенная схема(ы), затем 'pg_temp'.
    SET search_path = admin, pg_temp;
```

Эта функция должна обращаться к таблице admin.pwds, но без предложения SET или с предложением SET, включающим только admin, её можно «обмануть», создав временную таблицу pwds.

До PostgreSQL 8.3 предложение SET отсутствовало, так что старые функции могут содержать довольно сложную логику для сохранения, изменения и восстановления переменной search_path. Существующее теперь предложение SET позволяет сделать это намного проще.

Также следует помнить о том, что по умолчанию право выполнения для создаваемых функций имеет роль PUBLIC (за подробностями обратитесь к GRANT). Однако часто требуется разрешить доступ к функциям, работающим в контексте определившего, только некоторым пользователям. Для этого необходимо отозвать стандартные права PUBLIC и затем дать права на выполнение индивидуально. Чтобы не образовалось окно, в котором новая функция будет доступна всем, создайте её и назначьте права в одной транзакции. Например, так:

```
BEGIN;
CREATE FUNCTION check_password(uname TEXT, pass TEXT) ... SECURITY DEFINER;
REVOKE ALL ON FUNCTION check_password(uname TEXT, pass TEXT) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION check_password(uname TEXT, pass TEXT) TO admins;
COMMIT;
```

Совместимость

Команда CREATE FUNCTION определена в SQL:1999 и более поздних стандартах. Версия, реализованная в Postgres Pro, близка к стандартизированной, но соответствует ей не полностью. В частности, непереносимы атрибуты, а также различные языки реализаций.

Для совместимости с другими СУБД *режим_аргумента* можно записать после *имя_аргумента* или перед ним, но стандарту соответствует только первый вариант.

Для определения значений по умолчанию для параметров стандарт SQL поддерживает только синтаксис с ключевым словом DEFAULT. Синтаксис со знаком = используется в T-SQL и Firebird.

См. также

ALTER FUNCTION, DROP FUNCTION, GRANT, LOAD, REVOKE

CREATE GROUP

CREATE GROUP — создать роль в базе данных

Синтаксис

```
CREATE GROUP имя [ [ WITH ] параметр [ ... ] ]

Здесь параметр:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT предел_подключений
    | [ ENCRYPTED ] PASSWORD 'пароль'
    | VALID UNTIL 'дата_время'
    | IN ROLE имя_роли [, ...]
    | IN GROUP имя_роли [, ...]
    | ROLE имя_роли [, ...]
    | ADMIN имя_роли [, ...]
    | USER имя_роли [, ...]
    | SYSID uid
```

Описание

Оператор `CREATE GROUP` теперь является синонимом оператора CREATE ROLE.

Совместимость

Оператор `CREATE GROUP` отсутствует в стандарте SQL.

См. также

CREATE ROLE

CREATE INDEX

CREATE INDEX — создать индекс

Синтаксис

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] имя ] ON имя_таблицы
 [ USING метод ]
    ( { имя_столбца | ( выражение ) } [ COLLATE правило_сортировки ] [ класс_операторов
 ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
    [ INCLUDE ( имя_столбца [, ...] )]
    [ WITH ( параметр_хранения [= значение] [, ... ] ) ]
    [ TABLESPACE табл_пространство ]
    [ WHERE предикат ]
```

Описание

`CREATE INDEX` создаёт индексы по указанному столбцу(ам) заданного отношения, которым может быть таблица или материализованное представление. Индексы применяются в первую очередь для оптимизации производительности базы данных (хотя при неправильном использовании возможен и противоположный эффект).

Ключевое поле для индекса задаётся как имя столбца или выражение, заключённое в скобки. Если метод индекса поддерживает составные индексы, допускается указание нескольких полей.

Поле индекса может быть выражением, вычисляемым из значений одного или нескольких столбцов в строке таблицы. Это может быть полезно для получения быстрого доступа к данным по некоторому преобразованию исходных значений. Например, индекс, построенный по выражению `upper(col)`, позволит использовать поиск по индексу в предложении `WHERE upper(col) = 'JIM'`.

Postgres Pro предоставляет следующие методы индексов: B-дерево, хеш, GiST, SP-GiST, GIN и BRIN. Пользователи могут определить и собственные методы индексов, но это довольно сложная задача.

Если в команде присутствует предложение `WHERE`, она создаёт *частичный индекс*. Такой индекс содержит записи только для части таблицы, обычно более полезной для индексации, чем остальная таблица. Например, если таблица содержит информацию об оплаченных и неоплаченных счетах, при этом последних сравнительно немного, но именно эта часть таблицы наиболее востребована, то увеличить быстродействие можно, создав индекс только по этой части. Ещё одно возможное применение `WHERE` — добавив `UNIQUE`, обеспечить уникальность в подмножестве таблицы. Подробнее это рассматривается в Разделе 11.8.

Выражение в предложении `WHERE` может ссылаться только на столбцы нижележащей таблицы, но не обязательно ограничиваться теми, по которым строится индекс. В настоящее время в `WHERE` также нельзя использовать подзапросы и агрегатные выражения. Это же ограничение распространяется и на выражения в полях индексов.

Все функции и операторы, используемые в определении индекса, должны быть «постоянными», то есть, их результаты должны зависеть только от аргументов, но не от внешних факторов (например, содержимого другой таблицы или текущего времени). Это ограничение обеспечивает определённость поведения индекса. Чтобы использовать в выражении индекса или в предложении `WHERE` собственную функцию, не забудьте пометить её при создании как постоянную (IMMUTABLE).

Параметры

`UNIQUE`

Указывает, что система должна контролировать повторяющиеся значения в таблице при создании индекса (если в таблице уже есть данные) и при каждом добавлении данных. Попытки вставить или изменить данные, при которых будет нарушена уникальность индекса, будут завершаться ошибкой.

`CONCURRENTLY`

С этим указанием Postgres Pro построит индекс, не устанавливая никаких блокировок, которые бы предотвращали добавление, изменение или удаление записей в таблице, тогда как по умолчанию операция построения индекса блокирует запись (но не чтение) данных в таблице до своего завершения. С созданием индекса в этом режиме связан ряд особенностей, о которых следует знать, — см. Подраздел «Неблокирующее построение индексов».

Для временных таблиц `CREATE INDEX` всегда выполняется более простым, неблокирующим способом, так как они не могут использоваться никакими другими сеансами.

`IF NOT EXISTS`

Не считать ошибкой, если индекс с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующий индекс как-то соотносится с тем, который мог бы быть создан. Имя индекса является обязательным, когда указывается `IF NOT EXISTS`.

`INCLUDE`

Необязательное предложение `INCLUDE` позволяет указать список столбцов, которые войдут в индекс как *неключевые* столбцы. Неключевой столбец не может использоваться в условии поиска для сканирования по индексу, и он не учитывается при анализе ограничения уникальности или исключения, устанавливаемого индексом. Однако при сканировании только индекса содержимое неключевых столбцов может быть получено без обращения к целевой таблице, так как оно находится непосредственно в элементе индекса. Таким образом, в результате добавления неключевых столбцов сканирование только индекса может использоваться в тех запросах, где иначе оно было бы неприменимо.

Добавляя в индекс неключевые столбцы, особенно большого размера, есть смысл проявлять консерватизм. Если кортеж в индексе превышает максимально допустимый размер для данного типа индексов, вставить данные в таблицу не удастся. В неключевых столбцах дублируются данные из основной таблицы, что приводит к разрастанию индекса и может повлечь замедление запросов.

Для столбцов, перечисленных в `INCLUDE`, не нужны соответствующие классы операторов; это предложение может содержать столбцы, для типов данных которых не определены классы операторов для заданного метода доступа.

Выражения в неключевых столбцах не поддерживаются, так как их нельзя будет использовать при сканировании только индекса.

В настоящее время эту возможность поддерживает только метод доступа индекса-B-дерева. В таких индексах значения столбцов, указанных в предложении `INCLUDE`, включаются в кортежи на уровне листьев, которые соответствуют кортежам кучи, но не включаются в элементы верхних уровней, используемые для навигации в дереве.

`имя`

Имя создаваемого индекса. Указание схемы при этом не допускается; индекс всегда относится к той же схеме, что и родительская таблица. Если имя опущено, Postgres Pro формирует подходящее имя по имени родительской таблицы и именам индексируемых столбцов.

`имя_таблицы`

Имя индексируемой таблицы (возможно, дополненное схемой).

`метод`

Имя применяемого метода индекса. Возможные варианты: `btree`, `hash`, `gist`, `spgist`, `gin`, `brin` или установленный пользователем метод, например bloom. По умолчанию подразумевается метод `btree`.

имя_столбца

Имя столбца таблицы.

выражение

Выражение с одним или несколькими столбцами таблицы. Обычно выражение должно записываться в скобках, как показано в синтаксисе команды. Однако скобки можно опустить, если выражение записано в виде вызова функции.

правило_сортировки

Имя правила сортировки, применяемого для индекса. По умолчанию используется правило сортировки, заданное для индексируемого столбца, либо полученное для результата выражения индекса. Индексы с нестандартными правилами сортировки могут быть полезны для запросов, включающих выражения с такими правилами.

класс_операторов

Имя класса операторов. Подробнее об этом ниже.

ASC

Указывает порядок сортировки по возрастанию (подразумевается по умолчанию).

DESC

Указывает порядок сортировки по убыванию.

NULLS FIRST

Указывает, что значения NULL после сортировки оказываются перед остальными. Это поведение по умолчанию с порядком сортировки DESC.

NULLS LAST

Указывает, что значения NULL после сортировки оказываются после остальных. Это поведение по умолчанию с порядком сортировки ASC.

параметр_хранения

Имя специфичного для индекса параметра хранения. За подробностями обратитесь к Подразделу «Параметры хранения индекса».

табл_пространство

Табличное пространство, в котором будет создан индекс. Если не определено, выбирается default_tablespace, либо temp_tablespaces, при создании индекса временной таблицы.

предикат

Выражение ограничения для частичного индекса.

Параметры хранения индекса

Необязательное предложение WITH определяет *параметры хранения* для индекса. У каждого метода индекса есть свой набор допустимых параметров хранения. Следующий параметр принимают методы B-дерево, хеш, GiST и SP-GiST:

fillfactor

Фактор заполнения для индекса определяет в процентном отношении, насколько плотно метод индекса будет заполнять страницы индекса. Для B-деревьев концевые страницы заполняются до этого процента при начальном построении индекса и позже, при расширении индекса вправо (добавлении новых наибольших значений ключа). Если страницы впоследствии оказываются заполненными полностью, они будут разделены, что приводит к постепенному снижению эффективности индекса. Для B-деревьев по умолчанию используется фактор

заполнения 90, но его можно поменять на любое целое значение от 10 до 100. Фактор заполнения, равный 100, полезен для статических таблиц и помогает уменьшить физический размер таблицы, но для интенсивно изменяемых таблиц лучше использовать меньшее значение, чтобы разделять страницы приходилось реже. С другими методами индекса фактор заполнения действует по-другому, но примерно в том же ключе; значение фактора заполнения по умолчанию для разных методов разное.

Индексы GiST дополнительно принимают этот параметр:

`buffering`

Определяет, будет ли при построении индекса использоваться буферизация, описанная в Подразделе 58.4.1. Со значением `OFF` она отключена, с `ON` — включена, а с `AUTO` — отключена вначале, но может затем включиться на ходу, как только размер индекса достигнет значения effective_cache_size. По умолчанию подразумевается `AUTO`.

Индексы GIN принимают другие параметры:

`fastupdate`

Этот параметр управляет механизмом быстрого обновления, описанным в Подразделе 60.4.1. Он имеет логическое значение: `ON` включает быстрое обновление, `OFF` отключает его. (Другие возможные написания `ON` и `OFF` перечислены в Разделе 18.1.) Значение по умолчанию — `ON`.

> **Примечание**
>
> Выключение `fastupdate` в `ALTER INDEX` предотвращает помещение добавляемых в дальнейшем строк в список записей, ожидающих индексации, но записи, добавленные в этот список ранее, в нём остаются. Чтобы очистить очередь операций, надо затем выполнить `VACUUM` для этой таблицы или вызвать функцию `gin_clean_pending_list`.

`gin_pending_list_limit`

Пользовательский параметр gin_pending_list_limit. Его значение задаётся в килобайтах.

Индексы BRIN принимают другие параметры:

`pages_per_range`

Определяет, сколько блоков таблицы образуют зону блоков для каждой записи в индексе BRIN (за подробностями обратитесь к Разделу 61.1). Значение по умолчанию — `128`.

`autosummarize`

Определяет, будет ли производиться расчёт сводного значения для предыдущей зоны страниц, когда происходит добавление в следующую.

Неблокирующее построение индексов

Создание индекса может мешать обычной работе с базой данных. Обычно Postgres Pro блокирует запись в индексируемую таблицу и выполняет всю операцию построения индекса за одно сканирование таблицы. Другие транзакции могут продолжать читать таблицу, но при попытке вставить, изменить или удалить строки в таблице они будут заблокированы до завершения построения индекса. Это может оказать нежелательное влияние на работу производственной базы данных. Индексация очень больших таблиц может занимать много часов, и даже для маленьких таблиц построение индекса может заблокировать записывающие процессы на время, неприемлемое для производственной системы.

Postgres Pro поддерживает построение индексов без блокировки записи. Этот метод выбирается указанием `CONCURRENTLY` команды `CREATE INDEX`. Когда он используется, Postgres Pro должен выполнить два сканирования таблицы, а кроме того, должен дождаться завершения всех

существующих транзакций, которые потенциально могут модифицировать и использовать этот индекс. Таким образом, эта процедура требует проделать в сумме больше действий и выполняется значительно дольше, чем обычное построение индекса. Однако благодаря тому, что этот метод позволяет продолжать обычную работу с базой во время построения индекса, он оказывается полезным в производственной среде. Хотя разумеется, дополнительная нагрузка на процессор и подсистему ввода/вывода, создаваемая при построении индекса, может привести к замедлению других операций.

При неблокирующем построении индекса он попадает в системный каталог в одной транзакции, затем ещё два сканирования таблицы выполняются в двух других транзакциях. Перед каждым сканированием таблицы процедура построения индекса должна ждать завершения текущих транзакций, модифицировавших эту таблицу. После второго сканирования также необходимо дожидаться завершения всех транзакций, получивших снимок (см. Главу 13) перед вторым сканированием, включая транзакции, задействованные на любом этапе неблокирующего построения индексов для других таблиц. Наконец индекс может быть помечен как готовый к использованию, после чего команда `CREATE INDEX` завершается. Однако даже тогда индекс может быть не готов немедленно к применению в запросах: в худшем случае он не будет использоваться, пока существуют транзакции, начатые до начала построения индекса.

Если при сканировании таблицы возникает проблема, например взаимоблокировка или нарушение уникальности в уникальном индексе, команда `CREATE INDEX` завершится ошибкой, но оставит после себя «нерабочий» индекс. Этот индекс будет игнорироваться при чтении данных, так как он может быть неполным; однако с ним могут быть связаны дополнительные операции при изменениях. В psql встроенная команда `\d` помечает такой индекс как `INVALID`:

```
postgres=# \d tab
       Table "public.tab"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 col    | integer |           |          |
Indexes:
    "idx" btree (col) INVALID
```

Рекомендуемый в таких случаях способ исправления ситуации — удалить индекс и затем попытаться снова выполнить `CREATE INDEX CONCURRENTLY`. (Кроме того, можно перестроить его с помощью команды `REINDEX`. Но так как `REINDEX` не поддерживает неблокирующий режим, вряд ли этот вариант будет желательным.)

Ещё одна сложность, с которой можно столкнуться при неблокирующем построении уникального индекса, заключается в том, что ограничение уникальности уже влияет на другие транзакции, когда второе сканирование таблицы только начинается. Это значит, что нарушения ограничения могут проявляться в других запросах до того, как индекс становится доступным для использования и даже тогда, когда создать индекс в итоге не удаётся. Кроме того, если при втором сканировании происходит ошибка, «нерабочий» индекс оставляет в силе своё ограничение уникальности и дальше.

Метод неблокирующего построения поддерживает также индексы выражений и частичные индексы. Ошибки, произошедшие при вычислении этих выражений, могут привести к такому же поведению, как в вышеописанных случаях с нарушением ограничений уникальности.

Обычное построение индекса допускает параллельное построение других индексов для таблицы обычным методом, но неблокирующее построение для конкретной таблицы в один момент времени допускается только одно. Однако в любом случае никакие другие изменения схемы таблицы в это время не разрешаются. Другое отличие состоит в том, что в блоке транзакции может быть выполнена обычная команда `CREATE INDEX`, но не `CREATE INDEX CONCURRENTLY`.

Замечания

Информацию о том, когда могут применяться, и когда не применяются индексы, и в каких конкретных ситуациях они могут быть полезны, можно найти в Главе 11.

В настоящее время составные индексы поддерживаются только методами B-дерево, GiST, GIN и BRIN. По умолчанию такой индекс может включать до 32 полей. (Этот предел можно изменить, пересобрав Postgres Pro.) Уникальные индексы поддерживает только B-дерево.

Для каждого столбца индекса можно задать *класс операторов*. Этот класс определяет, какие операторы будут использоваться индексом для этого столбца. Например, индекс-B-дерево по четырёхбайтовым целым будет использовать класс `int4_ops`; этот класс операторов включает функции сравнения для таких значений. На практике обычно достаточно использовать класс операторов по умолчанию для типа данных столбца. Существование классов операторов объясняется в первую очередь тем, что для некоторых типов данных можно предложить более одного осмысленного порядка сортировки. Например, может возникнуть желание отсортировать комплексные числа как по абсолютному значению, так и по вещественной части. Это можно сделать, определив два класса операторов для типа данных и выбрав подходящий класс при создании индекса. За дополнительными сведениями о классах операторов обратитесь к Разделу 11.9 и Разделу 35.14.

Для методов индекса, поддерживающих сканирование по порядку (в настоящее время это поддерживает только B-дерево), можно изменить порядок сортировки индекса, добавив необязательные предложения `ASC`, `DESC`, `NULLS FIRST` или `NULLS LAST`. Так как упорядоченный индекс можно сканировать вперёд или назад, обычно не имеет смысла создавать индекс по убыванию (`DESC`) для одного столбца — этот порядок сортировки можно получить и с обычным индексом. Эти параметры имеют смысл при создании составных индексов так, что они будут соответствовать порядку сортировки, указанному в запросе со смешанным порядком, например `SELECT ... ORDER BY x ASC, y DESC`. Параметры `NULLS` полезны, когда требуется реализовать поведение «NULL внизу», изменив стандартное «NULL вверху», в запросах, зависящих от индексов, чтобы избежать дополнительной сортировки.

Система регулярно собирает статистику по всем столбцам таблицы. Новые индексы, построенные без применения выражений, могут эффективно использовать эту статистику сразу. Однако для создаваемых индексов по выражениям требуется выполнить `ANALYZE` или подождать, пока фоновый процесс автоочистки не проанализирует таблицу и не посчитает статистику для этих индексов.

Для большинства методов индексов скорость создания индекса зависит от значения maintenance_work_mem. Чем больше это значение, тем меньше времени требуется для создания индекса (если только заданное значение не превышает объём действительно доступной памяти, что влечёт за собой использование подкачки).

Для удаления индекса применяется DROP INDEX.

В предыдущих выпусках Postgres Pro также поддерживался метод индекса R-дерево. Сейчас он отсутствует, так как он не даёт значительных преимуществ по сравнению с GiST. Указание `USING rtree` команда `CREATE INDEX` будет интерпретировать как `USING gist`, для упрощения перевода старых баз на GiST.

Примеры

Создание уникального индекса-B-дерева по столбцу `title` в таблице `films`:

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

Создание уникального индекса-B-дерева по столбцу `title` с включёнными столбцами `director` и `rating` в таблице `films`:

```
CREATE UNIQUE INDEX title_idx ON films (title) INCLUDE (director, rating);
```

Создание индекса по выражению `lower(title)`, позволяющего эффективно выполнять регистронезависимый поиск:

```
CREATE INDEX ON films ((lower(title)));
```

(В этом примере мы решили опустить имя индекса, чтобы имя выбрала система, например `films_lower_idx`.)

Создание индекса с нестандартным правилом сортировки:

```
CREATE INDEX title_idx_german ON films (title COLLATE "de_DE");
```

Создание индекса с нестандартным порядком значений NULL:

```
CREATE INDEX title_idx_nulls_low ON films (title NULLS FIRST);
```

Создание индекса с нестандартным фактором заполнения:

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH (fillfactor = 70);
```

Создание индекса GIN с отключённым механизмом быстрого обновления:

```
CREATE INDEX gin_idx ON documents_table USING GIN (locations) WITH (fastupdate = off);
```

Создание индекса по столбцу `code` в таблице `films` и размещение его в табличном пространстве `indexspace`:

```
CREATE INDEX code_idx ON films (code) TABLESPACE indexspace;
```

Создание индекса GiST по координатам точек, позволяющего эффективно использовать операторы box с результатом функции преобразования:

```
CREATE INDEX pointloc
    ON points USING gist (box(location,location));
SELECT * FROM points
    WHERE box(location,location) && '(0,0),(1,1)'::box;
```

Создание индекса без блокировки записи в таблицу:

```
CREATE INDEX CONCURRENTLY sales_quantity_index ON sales_table (quantity);
```

Совместимость

`CREATE INDEX` является языковым расширением Postgres Pro. Средства обеспечения индексов в стандарте SQL не описаны.

См. также

ALTER INDEX, DROP INDEX

CREATE LANGUAGE

CREATE LANGUAGE — создать процедурный язык

Синтаксис

```
CREATE [ OR REPLACE ] [ PROCEDURAL ] LANGUAGE имя
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE имя
    HANDLER обработчик_вызова [ INLINE обработчик_внедрённого_кода ]
 [ VALIDATOR функция_проверки ]
```

Описание

`CREATE LANGUAGE` регистрирует в базе данных Postgres Pro новый процедурный язык. Впоследствии на этом языке можно будет определять новые функции и триггерные процедуры.

> **Примечание**
>
> Начиная с PostgreSQL 9.1, большинство процедурных языков были преобразованы в «расширения», так что они теперь устанавливаются посредством CREATE EXTENSION, а не `CREATE LANGUAGE`. Прямое использование `CREATE LANGUAGE` теперь следует ограничить скриптами установки расширений. Если в вашей базе данных есть «чистое» определение языка, возможно, полученное в результате обновления, его можно преобразовать в расширение, выполнив `CREATE EXTENSION имя_языка FROM unpackaged`.

`CREATE LANGUAGE` по сути связывает имя языка с функциями-обработчиками, исполняющими код, написанный на этом языке. За дополнительными сведениями о языковых обработчиках обратитесь к Главе 51.

Команда `CREATE LANGUAGE` имеет две формы. В первой форме пользователь задаёт только имя создаваемого языка, и сервер Postgres Pro получает подходящие параметры, обращаясь к системному каталогу `pg_pltemplate`. Во второй форме пользователь указывает параметры языка вместе с его именем. Вторая форма позволяет создать язык, не определённый заранее в `pg_pltemplate`, но этот подход считается устаревшим.

Когда сервер находит в каталоге `pg_pltemplate` запись для заданного имени языка, он будет использовать данные из каталога, даже если параметры языка указаны в команде. Это поведение упрощает загрузку экспортированных ранее файлов, которые, скорее всего, будут содержать устаревшую информацию о функциях поддержки языка.

Обычно, для того чтобы зарегистрировать новый язык, необходимо иметь право суперпользователя Postgres Pro. Однако владелец базы данных может зарегистрировать новый язык в этой базе данных, если язык определён в каталоге `pg_pltemplate` и помечен как допускающий создание владельцами БД (`tmpldbacreate` содержит true). По умолчанию доверенные языки могут создаваться владельцами базы данных, но это можно изменить, внеся коррективы в `pg_pltemplate`. Создатель языка становится его владельцем и может впоследствии удалить или переименовать его, либо назначить другого владельца.

`CREATE OR REPLACE LANGUAGE` создаст новый язык, либо заменит существующее определение. Если язык уже существует, его параметры изменяются в соответствии со значениями, указанными в команде или полученными из `pg_pltemplate`, при этом владелец языка и права доступа к нему не меняются, и все существующие функции, написанные на этом языке, по-прежнему считаются рабочими. Помимо обладания обычными правами, требующимися для создания языка, пользователь должен быть суперпользователем или владельцем существующего языка. Вариант с `REPLACE` в основном предназначен для случаев, когда язык уже существует. Если для языка есть запись в `pg_pltemplate`, то `REPLACE` фактически ничего не меняет в существующем определении,

кроме исключительного случая, когда запись в pg_pltemplate была изменена после создания языка.

Параметры

TRUSTED

Указание TRUSTED (доверенный) означает, что язык не даёт пользователю доступ к данным, к которым он не имел бы доступа без него. Если при регистрации языка это ключевое слово опущено, использовать этот язык для создания новых функций смогут только суперпользователи Postgres Pro.

PROCEDURAL

Это слово не несёт смысловой нагрузки.

имя

Имя процедурного языка. Оно должно быть уникальным среди всех языков в базе данных.

В целях обратной совместимости имя можно заключить в апострофы.

HANDLER *обработчик_вызова*

Здесь *обработчик_вызова* — имя ранее зарегистрированной функции, которая будет вызываться для исполнения процедур (функций) на этом языке. Обработчик вызова для процедурного языка должен быть написан на компилируемом языке, например, на C, с соглашениями о вызовах версии 1 и зарегистрирован в Postgres Pro как функция без аргументов, возвращающая фиктивный тип language_handler, который просто показывает, что эта функция — обработчик вызова.

INLINE *обработчик_внедрённого_кода*

Здесь *обработчик_внедрённого_кода* — имя ранее зарегистрированной функции, которая будет вызываться для выполнения анонимного блока кода (команды DO) на этом языке. Если *обработчик_внедрённого_кода* не определён, данный язык не будет поддерживать анонимные блоки кода. Этот обработчик должен принимать один аргумент типа internal, содержащий внутреннее представление команды DO, и обычно возвращает тип void. Значение, возвращаемое этим обработчиком, игнорируется.

VALIDATOR *функция_проверки*

Здесь *функция_проверки* — имя ранее зарегистрированной функции, которая будет вызываться при создании новой функции на этом языке и проверять её. Если функция проверки не задана, функции на этом языке при создании проверяться не будут. Функция проверки должна принимать один аргумент типа oid с идентификатором создаваемой функции и обычно возвращает void.

Функция проверки обычно проверяет тело создаваемой функции на синтаксические ошибки, но может также проанализировать и другие характеристики функции, например, поддержку определённых типов аргументов этим языком. Значение, возвращаемое этой функцией, игнорируется, об ошибках она должна сигнализировать, вызывая ereport().

Параметр TRUSTED и имена вспомогательных функций игнорируются, если на сервере для указанного языка имеется запись в pg_pltemplate.

Замечания

Для удаления процедурных языков следует использовать DROP LANGUAGE.

В системном каталоге pg_language (см. Раздел 49.29) записывается информация о языках, установленных в данный момент. Список установленных языков также показывает команда \dL в psql.

Чтобы создавать функции на процедурном языке, пользователь должен иметь право USAGE для этого языка. По умолчанию для доверенных языков право USAGE даётся роли PUBLIC (т. е. всем), однако при желании его можно отозвать.

Процедурные языки являются локальными по отношению к базам данных. Тем не менее, язык можно установить в базу данных template1, в результате чего он будет автоматически доступен во всех базах данных, создаваемых из неё впоследствии.

Обработчик вызова, обработчик встроенного кода (при наличии) и функция проверки (при наличии) должны уже существовать, если на сервере нет записи для этого языка в pg_pltemplate. Но если такая запись уже есть, функции могут не существовать — в случае отсутствия в базе данных они будут определены автоматически. (Это может привести к сбою в CREATE LANGUAGE, если в установленной системе не найдётся разделяемая библиотека, реализующая этот язык.)

В PostgreSQL до версии 7.3 обязательно требовалось объявить функции-обработчики, как возвращающие фиктивный тип opaque, а не language_handler. Для поддержки загрузки старых файлов экспорта БД, команда CREATE LANGUAGE принимает функции с объявленным типом результата opaque, но при этом выдаётся предупреждение и тип результата меняется на language_handler.

Примеры

Предпочитаемый способ создания любых процедурных языков довольно прост:

```
CREATE LANGUAGE plperl;
```

Для языка, не представленного в каталоге pg_pltemplate, требуется выполнить следующие действия:

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

Совместимость

CREATE LANGUAGE является расширением Postgres Pro.

См. также

ALTER LANGUAGE, CREATE FUNCTION, DROP LANGUAGE, GRANT, REVOKE

CREATE MATERIALIZED VIEW

CREATE MATERIALIZED VIEW — создать материализованное представление

Синтаксис

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] имя_таблицы
    [ (имя_столбца [, ...] ) ]
    [ WITH ( параметр_хранения [= значение] [, ... ] ) ]
    [ TABLESPACE табл_пространство ]
    AS запрос
    [ WITH [ NO ] DATA ]
```

Описание

`CREATE MATERIALIZED VIEW` определяет материализованное представление запроса. Заданный запрос выполняется и наполняет представление в момент вызова команды (если только не указано `WITH NO DATA`). Обновить представление позже можно, выполнив `REFRESH MATERIALIZED VIEW`.

Команда `CREATE MATERIALIZED VIEW` подобна `CREATE TABLE AS`, за исключением того, что она запоминает запрос, порождающий представление, так что это представление можно позже обновить по требованию. Материализованные представления сходны с таблицами во многом, но не во всём; например, не поддерживаются временные материализованные представления и автоматическая генерация OID.

Параметры

`IF NOT EXISTS`

Не считать ошибкой, если материализованное представление с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее материализованное представление как-то соотносится с тем, которое могло бы быть создано.

имя_таблицы

Имя создаваемого материализованного представления (возможно, дополненное схемой).

имя_столбца

Имя столбца в создаваемом материализованном представлении. Если имена столбцов не заданы явно, они определяются по именам столбцов результата запроса.

`WITH ( параметр_хранения [= значение] [, ... ] )`

Это предложение задаёт дополнительные параметры хранения для создаваемого материализованного представления; подробнее о них можно узнать в Подразделе «Параметры хранения». Все параметры, которые поддерживает `CREATE TABLE`, поддерживает и `CREATE MATERIALIZED VIEW`, за исключением `OIDS`. За дополнительной информацией обратитесь к CREATE TABLE.

`TABLESPACE табл_пространство`

Здесь *табл_пространство* — имя табличного пространства, в котором будет создано материализованное представление. Если оно не задано, выбирается default_tablespace.

запрос

Команда SELECT, TABLE или VALUES. Эта команда будет выполняться с ограничениями по безопасности; в частности, будут запрещены вызовы функций, которые сами создают временные таблицы.

`WITH [ NO ] DATA`

Это предложение указывает, будет ли материализованное представление наполняться в момент создания. Если материализованное представление не наполняется, оно помечается как нечитаемое, так что к нему нельзя будет обращаться до выполнения `REFRESH MATERIALIZED VIEW`.

Совместимость

`CREATE MATERIALIZED VIEW` является расширением Postgres Pro.

См. также

ALTER MATERIALIZED VIEW, CREATE TABLE AS, CREATE VIEW, DROP MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

CREATE OPERATOR

CREATE OPERATOR — создать оператор

Синтаксис

```
CREATE OPERATOR имя (
    PROCEDURE = имя_функции
    [, LEFTARG = тип_слева ] [, RIGHTARG = тип_справа ]
    [, COMMUTATOR = коммут_оператор ] [, NEGATOR = обратный_оператор ]
    [, RESTRICT = процедура_ограничения ] [, JOIN = процедура_соединения ]
    [, HASHES ] [, MERGES ]
)
```

Описание

`CREATE OPERATOR` определяет новый оператор, *имя*. Владельцем оператора становится пользователь, его создавший. Если указано имя схемы, оператор создаётся в ней, в противном случае — в текущей схеме.

Имя оператора образует последовательность из нескольких символов (не более чем `NAMEDATALEN`-1, по умолчанию 63) из следующего списка:

+ - * / < > = ~ ! @ # % ^ & | ` ?

Однако выбор имени ограничен ещё следующими условиями:

- Сочетания символов `--` и `/*` не могут присутствовать в имени оператора, так как они будут обозначать начало комментария.
- Многосимвольное имя оператора не может заканчиваться знаком `+` или `-`, если только оно не содержит также один из этих символов:

  ~ ! @ # % ^ & | ` ?

  Например, `@-` — допустимое имя оператора, а `*-` — нет. Благодаря этому ограничению, Postgres Pro может разбирать корректные SQL-запросы без пробелов между компонентами.
- Использование `=>` в качестве имени оператора считается устаревшим и может быть вовсе запрещено в будущих выпусках.

Оператор `!=` отображается в `<>` при вводе, так что эти два имени всегда равнозначны.

Необходимо определить либо `LEFTARG`, либо `RIGHTARG`, а для бинарных операторов оба аргумента. Для правых унарных операторов должен быть определён только `LEFTARG`, а для левых унарных — только `RIGHTARG`.

> **Примечание**
>
> Правые унарные, также называемые постфиксными, операторы признаны устаревшими и будут удалены в Postgres Pro версии 14.

Процедура *имя_функции* должна быть уже определена с помощью `CREATE FUNCTION` и иметь соответствующее число аргументов (один или два) указанных типов.

Другие предложения определяют дополнительные характеристики оптимизации. Их значение описано в Разделе 35.13.

Чтобы создать оператор, необходимо иметь право `USAGE` для типов аргументов и результата, а также право `EXECUTE` для нижележащей функции. Если указывается коммутативный или обратный оператор, нужно быть его владельцем.

Параметры

имя

Имя определяемого оператора. Допустимые в нём символы перечислены ниже. Указанное имя может быть дополнено схемой, например так: `CREATE OPERATOR myschema.+ (...)`. Если схема не указана, оператор создаётся в текущей схеме. При этом два оператора в одной схеме могут иметь одно имя, если они работают с разными типами данных. Такое определение операторов называется *перегрузкой*.

имя_функции

Функция, реализующая этот оператор.

тип_слева

Тип данных левого операнда оператора, если он есть. Этот параметр опускается для левых унарных операторов.

тип_справа

Тип данных правого операнда оператора, если он есть. Этот параметр опускается для правых унарных операторов.

коммут_оператор

Оператор, коммутирующий для данного.

обратный_оператор

Оператор, обратный для данного.

процедура_ограничения

Функция оценки избирательности ограничения для данного оператора.

процедура_соединения

Функция оценки избирательности соединения для этого оператора.

HASHES

Показывает, что этот оператор поддерживает соединение по хешу.

MERGES

Показывает, что этот оператор поддерживает соединение слиянием.

Чтобы задать имя оператора с указанием схемы в *коммут_оператор* или другом дополнительном аргументе, применяется синтаксис `OPERATOR()`, например:

```
COMMUTATOR = OPERATOR(myschema.===) ,
```

Замечания

За дополнительными сведениями обратитесь к Разделу 35.12.

Задать лексический приоритет оператора в команде `CREATE OPERATOR` невозможно, так как обработка приоритетов жёстко зашита в анализаторе. Подробнее приоритеты описаны в Подразделе 4.1.6.

Устаревшие параметры `SORT1`, `SORT2`, `LTCMP` и `GTCMP` ранее использовались для определения имён операторов сортировки, связанных с оператором, применяемым при соединении слиянием. Теперь это не требуется, так как информацию о связанных операторах теперь дают семейства операторов B-дерева. Если в команде отсутствует явное указание `MERGES`, все эти параметры игнорируются.

Для удаления пользовательских операторов из базы данных применяется DROP OPERATOR, а для изменения их свойств — ALTER OPERATOR.

Примеры

Следующая команда определяет новый оператор, проверяющий равенство площадей, для типа `box`:

```
CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    PROCEDURE = area_equal_procedure,
    COMMUTATOR = ===,
    NEGATOR = !==,
    RESTRICT = area_restriction_procedure,
    JOIN = area_join_procedure,
    HASHES, MERGES
);
```

Совместимость

`CREATE OPERATOR` является языковым расширением Postgres Pro. Средства определения пользовательских операторов в стандарте SQL не описаны.

См. также

ALTER OPERATOR, CREATE OPERATOR CLASS, DROP OPERATOR

CREATE OPERATOR CLASS

CREATE OPERATOR CLASS — создать класс операторов

Синтаксис

```
CREATE OPERATOR CLASS имя [ DEFAULT ] FOR TYPE тип_данных
  USING индексный_метод [ FAMILY имя_семейства ] AS
  {  OPERATOR номер_стратегии имя_оператора [ ( тип_операнда, тип_операнда ) ] [ FOR
 SEARCH | FOR ORDER BY семейство_сортировки ]
    | FUNCTION номер_опорной_функции [ ( тип_операнда [ , тип_операнда ] ) ] имя_функции
 ( тип_аргумента [, ...] )
    | STORAGE тип_хранения
  } [, ... ]
```

Описание

`CREATE OPERATOR CLASS` создаёт класс операторов. Класс операторов устанавливает, как данный тип будет использоваться в индексе, определяя, какие операторы исполняют конкретные роли или «стратегии» для этого типа данных и метода индекса. Также класс операторов определяет вспомогательные процедуры, которые будет задействовать метод индекса в случае выбора данного класса для столбца индекса. Все операторы и функции, используемые классом операторов, должны существовать до создания этого класса.

Если указывается имя схемы, класс операторов создаётся в указанной схеме, в противном случае — в текущей. Два класса операторов в одной схеме могут иметь одинаковые имена, только если они предназначены для разных методов индекса.

Владельцем класса операторов становится пользователь, создавший его. В настоящее время создавать классы операторов могут только суперпользователи. (Это ограничение введено потому, что ошибочное определение класса может вызвать нарушения или даже сбой в работе сервера.)

`CREATE OPERATOR CLASS` в настоящее время не проверяет, включает ли определение класса операторов все операторы и функции, требуемые для метода индекса, и образуют ли они целостный набор. Ответственность за правильность определения класса операторов лежит на пользователе.

Связанные классы операторов могут быть сгруппированы в *семейства операторов*. Чтобы поместить класс в существующее семейство, добавьте параметр `FAMILY` в `CREATE OPERATOR CLASS`. Без этого параметра новый класс помещается в семейство, имеющее то же имя, что и класс (если такое семейство не существует, оно создаётся).

За дополнительными сведениями обратитесь к Разделу 35.14.

Параметры

имя

Имя создаваемого класса операторов, возможно, дополненное схемой.

`DEFAULT`

Если присутствует это указание, класс операторов становится классом по умолчанию для своего типа данных. Для определённого типа данных и метода индекса можно определить не больше одного класса операторов по умолчанию.

тип_данных

Тип данных столбца, для которого предназначен этот класс операторов.

индексный_метод

Имя индексного метода, для которого предназначен этот класс операторов.

имя_семейства

Имя существующего семейства операторов, в которое будет добавлен этот класс. Если не указано, подразумевается семейство с тем же именем, что и класс (если такое семейство не существует, оно создаётся).

номер_стратегии

Номер стратегии индексного метода для оператора, связанного с данным классом операторов.

имя_оператора

Имя (возможно, дополненное схемой) оператора, связанного с данным классом операторов.

тип_операнда

В предложении `OPERATOR` это тип данных операнда, либо ключевое слово `NONE`, характеризующее левый унарный или правый унарный оператор. Типы операндов обычно можно опустить, когда они совпадают с типом данных класса операторов.

В предложении `FUNCTION` это тип данных операнда, который должна поддерживать эта функция, если он отличается от входного типа данных функции (для функций сравнения в B-деревьях и хеш-функций) или типа данных класса (для функций поддержки сортировки в B-деревьях и всех функций в классах операторов GiST, SP-GiST, GIN и BRIN). Обычно предполагаемые по умолчанию типы оказываются подходящими, так что *тип_операнда* указывать в `FUNCTION` не нужно (если это не функции сортировки B-дерева, которые предназначены для сравнения разных типов данных).

семейство_сортировки

Имя (возможно, дополненное схемой) существующего семейства операторов `btree`, описывающего порядок сортировки, связанный с оператором сортировки.

Если не указано ни `FOR SEARCH` (для поиска), ни `FOR ORDER BY` (для сортировки), подразумевается `FOR SEARCH`.

номер_опорной_функции

Номер опорной процедуры индексного метода для функции, связанной с данным классом операторов.

имя_функции

Имя (возможно, дополненное схемой) функции, которая является опорной процедурой индексного метода для данного класса операторов.

тип_аргумента

Тип данных параметра функции.

тип_хранения

Тип данных, фактически сохраняемых в индексе. Обычно это тип данных столбца, но некоторые методы индекса (в настоящее время, GiST, GIN и BRIN) могут работать с отличным от него типом. Предложение `STORAGE` может присутствовать, только если метод индекса позволяет использовать другой тип данных. Если *тип_данных* столбца задан как `anyarray`, *тип_хранения* может быть объявлен как `anyelement`, чтобы показать, что записи в индексе являются членами типа элемента, принадлежащего к фактическому типу массива, для которого создаётся конкретный индекс.

Предложения `OPERATOR`, `FUNCTION` и `STORAGE` могут указываться в любом порядке.

Замечания

Так как механизмы индексов не проверяют права доступа к функциям, прежде чем вызывать их, включение функций или операторов в класс операторов по сути даёт всем право на выполнение их. Обычно это не проблема для таких функций, какие бывают полезны в классе операторов.

Операторы не должны реализовываться в функциях на языке SQL. SQL-функция вероятнее всего будет встроена в вызывающий запрос, что помешает оптимизатору понять, что этот запрос соответствует индексу.

До PostgreSQL 8.4 предложение `OPERATOR` могло включать указание `RECHECK`. Теперь это не поддерживается, так как оператор индекса может быть «неточным» и это определяется на ходу в момент выполнения. Это позволяет эффективно справляться с ситуациями, когда оператор может быть или не быть неточным.

Примеры

Команда в следующем примере определяет класс операторов индекса GiST для типа данных `_int4` (массива из `int4`). Полный пример приведён в модуле intarray.

```
CREATE OPERATOR CLASS gist__int_ops
    DEFAULT FOR TYPE _int4 USING gist AS
        OPERATOR        3       &&,
        OPERATOR        6       = (anyarray, anyarray),
        OPERATOR        7       @>,
        OPERATOR        8       <@,
        OPERATOR        20      @@ (_int4, query_int),
        FUNCTION        1       g_int_consistent (internal, _int4, smallint, oid,
 internal),
        FUNCTION        2       g_int_union (internal, internal),
        FUNCTION        3       g_int_compress (internal),
        FUNCTION        4       g_int_decompress (internal),
        FUNCTION        5       g_int_penalty (internal, internal, internal),
        FUNCTION        6       g_int_picksplit (internal, internal),
        FUNCTION        7       g_int_same (_int4, _int4, internal);
```

Совместимость

`CREATE OPERATOR CLASS` является расширением Postgres Pro. Команда `CREATE OPERATOR CLASS` отсутствует в стандарте SQL.

См. также

ALTER OPERATOR CLASS, DROP OPERATOR CLASS, CREATE OPERATOR FAMILY, ALTER OPERATOR FAMILY

CREATE OPERATOR FAMILY

CREATE OPERATOR FAMILY — создать семейство операторов

Синтаксис

```
CREATE OPERATOR FAMILY имя USING индексный_метод
```

Описание

`CREATE OPERATOR FAMILY` создаёт новое семейство операторов. Семейство операторов определяет коллекцию связанных классов операторов и, возможно, некоторые дополнительные операторы и вспомогательные функции, совместимые с этими классами, но не требующиеся для функционирования каких-либо индексов. (Операторы и функции, требующиеся для работы индексов, следует группировать в соответствующем классе операторов, а не «слабо» связывать в семействе операторов. Обычно операторы с операндами одного типа привязываются к классам операторов, тогда как операторы для двух разных типов могут быть слабо связаны в семействе, содержащем классы операторов для обоих типов.)

Создаваемое семейство операторов изначально не содержит ничего. Оно должно наполняться последующими командами `CREATE OPERATOR CLASS`, добавляющими в него классы операторов, и, возможно, командами `ALTER OPERATOR FAMILY`, добавляющими «слабосвязанные» операторы и соответствующие вспомогательные функции.

Если указывается имя схемы, семейство операторов создаётся в указанной схеме, в противном случае — в текущей. Два семейства операторов в одной схеме могут иметь одинаковые имена, только если они предназначены для разных методов индекса.

Владельцем семейства операторов становится пользователь, создавший его. В настоящее время создавать семейства операторов могут только суперпользователи. (Это ограничение введено потому, что ошибочное определение семейства может вызвать нарушения или даже сбой в работе сервера.)

За дополнительными сведениями обратитесь к Разделу 35.14.

Параметры

имя

Имя создаваемого семейства операторов, возможно, дополненное схемой.

индексный_метод

Имя индексного метода, для которого предназначено это семейство операторов.

Совместимость

`CREATE OPERATOR FAMILY` является расширением Postgres Pro. Команда `CREATE OPERATOR FAMILY` отсутствует в стандарте SQL.

См. также

ALTER OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS

CREATE POLICY

CREATE POLICY — создать новую политику защиты на уровне строк для таблицы

Синтаксис

```
CREATE POLICY имя ON имя_таблицы
    [ AS { PERMISSIVE | RESTRICTIVE } ]
    [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
    [ TO { имя_роли | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( выражение_USING ) ]
    [ WITH CHECK ( выражение_CHECK ) ]
```

Описание

Команда `CREATE POLICY` определяет для таблицы новую политику защиты на уровне строк. Заметьте, что для таблицы должна быть включена защита на уровне строк (using `ALTER TABLE ... ENABLE ROW LEVEL SECURITY`), чтобы созданные политики действовали.

Политика даёт разрешение на выборку, добавление, изменение или удаление строк, удовлетворяющих соответствующему выражению политики. Существующие строки таблицы проверяются по выражению, указанному в `USING`, тогда как строки, которые могут быть созданы командами `INSERT` или `UPDATE` проверяются по выражению, указанному в `WITH CHECK`. Когда выражение `USING` истинно для заданной строки, эта строка видна пользователю, а если ложно или выдаёт NULL, строка не видна. Когда выражение `WITH CHECK` истинно для заданной строки, эта строка добавляется или изменяется, а если ложно или выдаёт NULL, происходит ошибка.

Для операторов `INSERT` и `UPDATE` выражения `WITH CHECK` применяются после срабатывания триггеров `BEFORE`, но до того, как будут собственно модифицированы данные. Таким образом, триггер `BEFORE ROW` может изменить данные, подлежащие добавлению, и повлиять на результат условия политики защиты. Выражения `WITH CHECK` обрабатываются до каких-либо других ограничений.

Имена политик задаются на уровне таблицы. Таким образом, одно имя политики можно использовать в нескольких разных таблицах и в каждой дать отдельное, подходящее этой таблице определение политики.

Политики могут применяться для определённых команд или для определённых ролей. По умолчанию создаваемые политики применяются для всех команд и ролей, если явно не задано другое. К одной команде могут применяться несколько политик; подробнее рассказывается ниже. В Таблице 243 показано, как к определённым командам применяются разные типы политик.

Для политик, которые могут иметь и выражения `USING`, и выражения `WITH CHECK` (`ALL` и `UPDATE`), в случае отсутствия выражения `WITH CHECK` выражение `USING` будет использоваться и для определения видимости строк (обычное назначение `USING`) и для определения, какие строки разрешено добавить (назначение `WITH CHECK`).

Если для таблицы включена защита на уровне строк, но применимые политики отсутствуют, предполагается политика «запрета по умолчанию», так что никакие строки нельзя будет увидеть или изменить.

Параметры

имя

Имя создаваемой политики. Оно должно отличаться от имён других политик для этой таблицы.

имя_таблицы

Имя (возможно, дополненное схемой) существующей таблицы (или представления), к которой применяется эта политика.

PERMISSIVE

Указывает, что создаваемая политика должна быть разрешительной. Все разрешительные политики, которые применяются к данному запросу, будут объединяться вместе логическим оператором «ИЛИ». Создавая разрешительные политики, администраторы могут расширять множество записей, к которым можно обращаться. Политики являются разрешительными по умолчанию.

RESTRICTIVE

Указывает, что создаваемая политика должна быть ограничительной. Все ограничительные политики, которые применяются к данному запросу, будут объединяться вместе логическим оператором «И». Создавая ограничительные политики, администраторы могут сократить множество записей, к которым можно обращаться, так как для каждой записи должны удовлетворяться все ограничительные политики.

Заметьте, что для получения доступа к записям должна быть определена минимум одна разрешительная политика, и только в дополнение к ней могут быть определены имеющие смысл ограничительные политики, ограничивающие доступ. Если разрешительные политики отсутствуют, ни к каким записям обращаться нельзя. Когда определены и разрешительные, и ограничительные политики, запись будет доступна, если удовлетворяется минимум одна из разрешительных политик и все ограничительные.

команда

Команда, к которой применяется политика. Допустимые варианты: ALL, SELECT, INSERT, UPDATE и DELETE. ALL (все) подразумевается по умолчанию. Особенности их применения описаны ниже.

имя_роли

Роль (роли), к которой применяется политика. По умолчанию подразумевается PUBLIC, то есть политика применяется ко всем ролям.

выражение_USING

Произвольное условное выражение SQL (возвращающее boolean). Это условное выражение не может содержать агрегатные или оконные функции. Когда включена защита на уровне строк, оно добавляется в запросы, обращающиеся к данной таблице, и в их результатах оказываются видимыми только те строки, для которых оно выдаёт true. Все строки, для которых это выражение возвращает false или NULL, не будут видны пользователю (в запросе SELECT), и не будут доступны для модификации (запросами UPDATE или DELETE). Такая строка просто пропускается, ошибка при этом не выдаётся.

выражение_CHECK

Произвольное условное выражение SQL (возвращающее boolean). Это условное выражение не может содержать агрегатные или оконные функции. Когда включена защита на уровне строк, оно применяется в запросах INSERT и UPDATE к этой таблице, так что в них принимаются только те строки, для которых оно выдаёт true. Если это выражение выдаёт false или NULL для любой из добавляемых записей или записей, получаемых при изменении, выдаётся ошибка. Заметьте, что *ограничение_проверки* вычисляется для предлагаемого нового содержимого строки, а не для существующих данных.

Политики по командам

ALL

Указание ALL для политики означает, что она применяется ко всем командам, вне зависимости от типа. Если существует политика ALL и другие более детализированные политики, тогда

будет применяться и политика ALL, и более детализированная политика (или политики). Кроме того, политики ALL с выражением USING будут применяться и к стороне выборки, и к стороне изменения данных в запросе, если определено только выражение USING.

Например, когда выполняется UPDATE, политика ALL будет фильтровать и строки, которые сможет прочитать UPDATE для изменения (применяя выражение USING), и окончательные изменённые строки, проверяя, можно ли записать их в таблицу (применяя выражение WITH CHECK, если оно определено, или USING в противном случае). Если команда INSERT или UPDATE пытается добавить в таблицу строки, не удовлетворяющие выражению WITH CHECK политики ALL, вся команда будет прервана.

SELECT

Указание SELECT для политики означает, что она применяется к запросам SELECT и тогда, когда при обращении к отношению, для которого определена политика, задействуется право SELECT. В результате запрос SELECT выдаст только те записи из отношения, которые удовлетворят политике SELECT, и запрос, использующий право SELECT, например, запрос UPDATE, увидит только записи, разрешённые политикой SELECT. Для политики SELECT не может задаваться выражение WITH CHECK, так как оно действует только когда записи читаются из отношения.

INSERT

Указание INSERT для политики означает, что она применяется к командам INSERT. Если вставляемые строки не проходят проверку политики, выдаётся ошибка нарушения политики и вся команда INSERT прерывается. Для политики INSERT не может задаваться выражение USING, так как она действует только когда в отношение добавляются записи.

Заметьте, что INSERT с указанием ON CONFLICT DO UPDATE проверяет выражения WITH CHECK политик INSERT только для строк, добавляемых в отношение по пути INSERT.

UPDATE

Выбор типа UPDATE для политики означает, что она будет применяться к командам UPDATE, SELECT FOR UPDATE и SELECT FOR SHARE, а также к дополнительным предложениям ON CONFLICT DO UPDATE команд INSERT. Так как UPDATE подразумевает извлечение существующей записи и замену её новой изменённой записью, политики UPDATE принимают как выражение USING, так и WITH CHECK. Выражение USING определяет, какие записи команда UPDATE сможет увидеть для последующего изменения, а выражение WITH CHECK — какие изменённые строки сохранить в отношении.

Если в какой-либо строке изменённые значения не будут удовлетворять выражению WITH CHECK, произойдёт ошибка и вся команда будет прервана. Если указывается только предложение USING, его выражение будет применяться и в качестве USING, и в качестве выражения WITH CHECK.

Обычно команде UPDATE также нужно прочитать данные из столбцов подлежащего изменению отношения (например, в предложении WHERE или RETURNING либо в выражении в правой части предложения SET). В этом случае также требуется иметь права SELECT в изменяемом отношении и в дополнение к политикам UPDATE будут применяться соответствующие политики SELECT или ALL. Таким образом, помимо того, что пользователю должны разрешать изменение строк политики UPDATE или ALL, ему также должны разрешать доступ к изменяемым строкам политики SELECT или ALL.

Когда для команды INSERT задано вспомогательное предложение ON CONFLICT DO UPDATE, если выбирается путь UPDATE, строка, подлежащая изменению, сначала проверяется по выражениям USING всех политик UPDATE, а затем изменённая строка ещё раз проверяется по выражениям WITH CHECK. Заметьте, однако, что в отличие от отдельной команды UPDATE, если существующая строка не удовлетворяет выражениям USING, будет выдана ошибка (путь UPDATE *никогда* не пропускается неявно).

DELETE

Указание DELETE для политики означает, что она применяется к командам DELETE. Команда DELETE будет видеть только те строки, которые позволит эта политика. При этом строки могут быть видны через SELECT, но удалить их будет нельзя, если они не удовлетворяют выражению USING политики DELETE.

В большинстве случаев команде DELETE также нужно прочитать данные из столбцов в отношении, из которого осуществляется удаление (например, в предложении WHERE или RETURNING). В таких случаях необходимо также иметь право SELECT для этого отношения, и в дополнение к политикам DELETE будут применятся соответствующие политики SELECT или ALL. Таким образом, пользователь должен получить доступ к удаляемым строкам через политики SELECT или ALL, помимо того что удаление этих строк ему должны разрешить политики DELETE или ALL.

Для политики DELETE не может задаваться выражение WITH CHECK, так как она применяется только тогда, когда записи удаляются из отношения, а в этом случае новые строки, подлежащие проверке, отсутствуют.

Таблица 243. Политики, применяемые для разных команд

Команда	Политика SELECT/ALL	Политика INSERT/ALL	Политика UPDATE/ALL		Политика DELETE/ALL
	Выражение USING	Выражение WITH CHECK	Выражение USING	Выражение WITH CHECK	Выражение USING
SELECT	Существующая строка	—	—	—	—
SELECT FOR UPDATE/SHARE	Существующая строка	—	Существующая строка	—	—
INSERT	—	Новая строка	—	—	—
INSERT ... RETURNING	Новая строка [a]	Новая строка	—	—	—
UPDATE	Существующие и новые строки [a]	—	Существующая строка	Новая строка	—
DELETE	Существующая строка [a]	—	—	—	Существующая строка
ON CONFLICT DO UPDATE	Существующие и новые строки	—	Существующая строка	Новая строка	—

[a] Если для существующей или новой строки требуется доступ на чтение (например, предложение WHERE или RETURNING, обращающееся к столбцам отношения).

Применение нескольких политик

Когда к одной команде применяются несколько политик для различных типов команд (как например, политики SELECT и UPDATE применяются к команде UPDATE), пользователь должен иметь разрешения всех этих типов (например, разрешение для выборки строк из отношения, а также разрешение на их изменение). Таким образом, выражения для одного типа политики комбинируются с выражениями для другого типа операцией И.

Когда к одной команде применяются несколько политик для одного типа команды, доступ к отношению должна дать как минимум одна разрешительная (PERMISSIVE) политика, а также должны удовлетворяться все ограничительные (RESTRICTIVE) политики. Таким образом выражения всех политик PERMISSIVE объединяются операцией ИЛИ, выражения всех политик RESTRICTIVE объединяются операцией И, а полученные результаты объединяются операцией И. Если политики PERMISSIVE отсутствуют, доступ запрещается.

Заметьте, что при объединении нескольких политик, политики ALL применяются как политики каждого применимого в данном случае типа.

Например, в команде UPDATE, требующей разрешений и для SELECT, и для UPDATE, в случае существования нескольких применимых политик каждого типа они будут объединяться следующим образом:

```
выражение from RESTRICTIVE SELECT/ALL policy 1
AND
выражение from RESTRICTIVE SELECT/ALL policy 2
AND
...
AND
(
  выражение from PERMISSIVE SELECT/ALL policy 1
  OR
  выражение from PERMISSIVE SELECT/ALL policy 2
  OR
  ...
)
AND
выражение from RESTRICTIVE UPDATE/ALL policy 1
AND
выражение from RESTRICTIVE UPDATE/ALL policy 2
AND
...
AND
(
  выражение from PERMISSIVE UPDATE/ALL policy 1
  OR
  выражение from PERMISSIVE UPDATE/ALL policy 2
  OR
  ...
)
```

Замечания

Чтобы создать или изменить политики для таблицы, нужно быть её владельцем.

Хотя политики применяются к явно выполняемым запросам к таблицам БД, они не применяются, когда система выполняет внутренние проверки ссылочной целостности или проверяет ограничения. Это означает, что существуют косвенные пути проверить существование заданного значения. Например, можно попытаться вставить повторяющееся значение в столбец, образующий первичный ключ или имеющую ограничение уникальности. Если при этом произойдёт ошибка, пользователь может заключить, что это значение уже существует. (В данном случае предполагается, что политика разрешает пользователю вставлять записи, которые он может не видеть.) Подобный приём также возможен, если пользователь может вставлять записи в таблицу, которая ссылается на другую, иным образом не видимую. Существование значения можно определить, вставив его в подчинённую таблицу, при этом успешный результат операции будет признаком того, что это значение есть в главной таблице. Эти изъяны можно устранить, либо тщательно разработав политики, которые вовсе не позволят пользователям выполнять операции добавления, изменения и удаления, по результатам которых можно узнать о значениях в таблицах, не видимых иным образом, либо используя генерируемые значения (например, суррогатные ключи).

Вообще система будет применять фильтры, устанавливаемые политиками безопасности, до условий в запросах пользователя, чтобы предотвратить нежелательную утечку защищаемых данных через пользовательские функции, которые могут быть недоверенными. Однако функции и операторы, помеченные системой (или системным администратором) как LEAKPROOF (герметичные) могут вычисляться до условий политики, так как они считаются доверенными.

Так как выражения политики добавляются непосредственно в запрос пользователя, они выполняются с правами пользователя, запускающего исходный запрос. Таким образом, пользователи, на которых распространяется заданная политика, должны иметь права для обращения ко всем таблицам и функциям, задействованным в выражении, иначе им просто будет отказано в доступе при попытке обращения к целевой таблице (если для неё включена защита на уровне строк). Однако это не влияет на работу представлений — как и с обычными запросами и представлениями, проверки разрешений и политики для нижележащих таблиц представления будут выполняться с правами владельца представления, и при этом будут действовать политики, распространяющиеся на этого владельца.

Дополнительное описание и практические примеры можно найти в Разделе 5.7.

Совместимость

`CREATE POLICY` является расширением Postgres Pro.

См. также

ALTER POLICY, DROP POLICY, ALTER TABLE

CREATE PUBLICATION

CREATE PUBLICATION — создать публикацию

Синтаксис

```
CREATE PUBLICATION имя
    [ FOR TABLE [ ONLY ] имя_таблицы [ * ] [, ...]
      | FOR ALL TABLES ]
    [ WITH ( параметр_публикации [= значение] [, ... ] ) ]
```

Описание

`CREATE PUBLICATION` создаёт новую публикацию в текущей базе данных. Имя публикации должно отличаться от имён других существующих публикаций в текущей базе.

Публикация по сути является группой таблиц, изменения в данных которых должны реплицироваться с использованием логической репликации. Подробнее о том, как публикации вписываются в схему логической репликации, рассказывается в Разделе 30.1.

Параметры

имя

Имя новой публикации.

`FOR TABLE`

Задаёт список таблиц, добавляемых в публикацию. Если перед именем таблицы указано `ONLY`, в публикацию добавляется только заданная таблица. Без `ONLY` добавляется и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно добавить необязательное указание `*`, чтобы явно обозначить, что должны включаться и все дочерние таблицы.

В публикацию могут включаться только постоянные базовые таблицы. Временные, нежурналируемые, сторонние и секционированные таблицы, а также материализованные и обычные представления не могут входить в публикацию. Для реплицирования секционированной таблицы в публикацию нужно добавить её отдельные секции.

`FOR ALL TABLES`

Устанавливает, что данная публикация охватывает изменения во всех таблицах в базе данных, включая таблицы, которые будут созданы позже.

`WITH ( параметр_публикации [= значение] [, ... ] )`

В этом предложении могут задаваться следующие необязательные параметры публикации:

`publish` (`string`)

Этот параметр определяет, какие операции DML будет передавать новая публикация её подписчикам. В качестве его значения через запятую задаётся список операций из следующих: `insert`, `update` и `delete`. По умолчанию публикуются все действия, так что этот параметр имеет значение по умолчанию `'insert, update, delete'`.

Замечания

Если не задано ни `FOR TABLE`, ни `FOR ALL TABLES`, публикация создаётся с пустым набором таблиц. Это полезно, если таблицы будут добавляться позднее.

Создание публикации не влечёт немедленный запуск репликации. Эта операция только определяет логику группирования и фильтрации для будущих подписчиков.

Чтобы создать публикацию, пользователь должен иметь право CREATE в текущей базе данных. (Разумеется, на суперпользователей это условие не распространяется.)

Чтобы добавить таблицу в публикацию, пользователь должен иметь права владельца этой таблицы. Для использования предложения FOR ALL TABLES пользователь должен быть суперпользователем.

Таблицы, добавляемые в публикацию, которая охватывает операции UPDATE и/или DELETE, должны иметь свойство REPLICA IDENTITY. В противном случае отслеживание этих операций для таблиц будет запрещено.

Для команды INSERT ... ON CONFLICT публикация будет выдавать операцию, к которой сводится команда. В зависимости от исхода команды, она может быть опубликована либо как INSERT, либо как UPDATE, либо не будет опубликована вовсе.

Команды COPY ... FROM публикуются в виде операций INSERT.

Команда TRUNCATE и операции DDL не публикуются.

Примеры

Создание публикации, охватывающей изменения в двух таблицах:

```
CREATE PUBLICATION mypublication FOR TABLE users, departments;
```

Создание публикации, охватывающей все изменения во всех таблицах:

```
CREATE PUBLICATION alltables FOR ALL TABLES;
```

Создание публикации, охватывающей только операции INSERT в одной таблице:

```
CREATE PUBLICATION insert_only FOR TABLE mydata
    WITH (publish = 'insert');
```

Совместимость

CREATE PUBLICATION является расширением Postgres Pro.

См. также

ALTER PUBLICATION, DROP PUBLICATION, CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

CREATE ROLE

CREATE ROLE — создать роль в базе данных

Синтаксис

```
CREATE ROLE имя [ [ WITH ] параметр [ ... ] ]

Здесь параметр:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT предел_подключений
    | [ ENCRYPTED ] PASSWORD 'пароль'
    | VALID UNTIL 'дата_время'
    | IN ROLE имя_роли [, ...]
    | IN GROUP имя_роли [, ...]
    | ROLE имя_роли [, ...]
    | ADMIN имя_роли [, ...]
    | USER имя_роли [, ...]
    | SYSID uid
```

Описание

`CREATE ROLE` добавляет новую роль в кластер баз данных Postgres Pro. Роль — это сущность, которая может владеть объектами и иметь определённые права в базе; роль может представлять «пользователя», «группу» или и то, и другое, в зависимости от варианта использования. За информацией об управлении пользователями и проверке подлинности обратитесь к Главе 20 и Главе 19. Чтобы выполнить эту команду, необходимо быть суперпользователем или иметь право `CREATEROLE`.

Учтите, что роли определяются на уровне кластера баз данных, так что они действуют во всех базах в кластере.

Параметры

имя

Имя создаваемой роли.

`SUPERUSER`
`NOSUPERUSER`

Эти предложения определяют, будет ли эта роль «суперпользователем», который может переопределить все ограничения доступа в базе данных. Статус суперпользователя несёт опасность и назначать его следует только в случае необходимости. Создать нового суперпользователя может только суперпользователь. В отсутствие этих предложений по умолчанию подразумевается `NOSUPERUSER`.

`CREATEDB`
`NOCREATEDB`

Эти предложения определяют, сможет ли роль создавать базы данных. Указание `CREATEDB` даёт новой роли это право, а `NOCREATEDB` запрещает роли создавать базы данных. По умолчанию подразумевается `NOCREATEDB`.

`CREATEROLE`
`NOCREATEROLE`

Эти предложения определяют, сможет ли роль создавать новые роли (т. е. выполнять `CREATE ROLE`). Роль с правом `CREATEROLE` может также изменять и удалять другие роли. По умолчанию подразумевается `NOCREATEROLE`.

`INHERIT`
`NOINHERIT`

Эти предложения определяют, будет ли роль «наследовать» права ролей, членом которых она является. Роль с атрибутом `INHERIT` может автоматически использовать в базе данных любые права, назначенные всем ролям, в которые она включена, непосредственно или опосредованно. Без `INHERIT` членство в другой роли позволяет только выполнить `SET ROLE` и переключиться на эту роль; правами, назначенными другой роли, можно будет пользоваться только после этого. По умолчанию подразумевается `INHERIT`.

`LOGIN`
`NOLOGIN`

Эти предложения определяют, разрешается ли новой роли вход на сервер; то есть, может ли эта роль стать начальным авторизованным именем при подключении клиента. Можно считать, что роль с атрибутом `LOGIN` соответствует пользователю. Роли без этого атрибута бывают полезны для управления доступом в базе данных, но это не пользователи в обычном понимании. По умолчанию подразумевается вариант `NOLOGIN`, за исключением вызова `CREATE ROLE` в виде CREATE USER.

`REPLICATION`
`NOREPLICATION`

Эти предложения определяют, будет ли роль ролью репликации. Чтобы роль могла подключаться к серверу в режиме репликации (в режиме физической или логической репликации) и создавать/удалять слоты репликации, у неё должен быть этот атрибут (либо это должна быть роль суперпользователя). Роль, имеющая атрибут `REPLICATION`, обладает очень большими привилегиями и поэтому этот атрибут должны иметь только роли, фактически используемые для репликации. По умолчанию подразумевается вариант `NOREPLICATION`. Создавать роли с атрибутом `REPLICATION` разрешено только суперпользователям.

`BYPASSRLS`
`NOBYPASSRLS`

Эти предложения определяют, будут ли для роли игнорироваться все политики защиты на уровне строк (RLS). По умолчанию подразумевается вариант `NOBYPASSRLS`. Создавать роли с атрибутом `NOBYPASSRLS` разрешено только суперпользователям.

Заметьте, что pg_dump по умолчанию отключает `row_security` (устанавливает значение `OFF`), чтобы гарантированно было выгружено всё содержимое таблицы. Если пользователь, запускающий pg_dump, не будет иметь необходимых прав, он получит ошибку. Однако суперпользователи и владелец выгружаемой таблицы всегда обходят защиту RLS.

`CONNECTION LIMIT` *`предел_подключений`*

Если роли разрешён вход, этот параметр определяет, сколько параллельных подключений может установить роль. Значение -1 (по умолчанию) снимает ограничение. Заметьте, что под это ограничение подпадают только обычные подключения. Ни подготовленные транзакции, ни соединения фоновых рабочих процессов в расчёт не берутся.

[`ENCRYPTED`] `PASSWORD` *`пароль`*

Задаёт пароль роли. (Пароль полезен только для ролей с атрибутом `LOGIN`, но задать его можно и для ролей без такого атрибута.) Если проверка подлинности по паролю не будет

использоваться, этот параметр можно опустить. При указании пустого значения будет задан пароль NULL, что не позволит данному пользователю пройти проверку подлинности по паролю. При желании пароль NULL можно установить явно, указав PASSWORD NULL.

> **Примечание**
>
> Если задана пустая строка, пароль также будет сброшен в NULL, но в Postgres Pro до версии 10 было не так. В ранних версиях пустая строка могла приниматься или нет, в зависимости от метода аутентификации и версии сервера, а libpq не давала использовать такой пароль в любом случае. Во избежание неоднозначности указывать пустую строку в качестве пароля не следует.

Пароль всегда хранится в системных каталогах в зашифрованном виде. Ключевое слово ENCRYPTED не имеет эффекта, но принимается для обратной совместимости. Метод шифрования определяется параметром конфигурации password_encryption. Если представленная строка пароля уже зашифрована с применением MD5 или SCRAM, она сохраняется как есть вне зависимости от значения password_encryption (так как система не может расшифровать переданный зашифрованной пароль, чтобы зашифровать его по другому алгоритму). Это позволяет пересохранять зашифрованные пароли при выгрузке/восстановлении.

VALID UNTIL '*дата_время*'

Предложение VALID UNTIL устанавливает дату и время, после которого пароль роли перестаёт действовать. Если это предложение отсутствует, срок действия пароля будет неограниченным.

IN ROLE *имя_роли*

В предложении IN ROLE перечисляются одна или несколько существующих ролей, в которые будет немедленно включена новая роль. (Заметьте, что добавить новую роль с правами администратора таким образом нельзя; для этого надо отдельно выполнить команду GRANT.)

IN GROUP *имя_роли*

IN GROUP — устаревшее написание предложения IN ROLE.

ROLE *имя_роли*

В предложении ROLE перечисляются одна или несколько существующих ролей, которые автоматически становятся членами создаваемой роли. (По сути таким образом новая роль становится «группой».)

ADMIN *имя_роли*

Предложение ADMIN подобно ROLE, но перечисленные в нём роли включаются в новую роль с атрибутом WITH ADMIN OPTION, что даёт им право включать в эту роль другие роли.

USER *имя_роли*

Предложение USER является устаревшим написанием предложения ROLE.

SYSID *uid*

Предложение SYSID игнорируется, но принимается для обратной совместимости.

Замечания

Для изменения атрибутов роли применяется ALTER ROLE, а для удаления роли — DROP ROLE. Все атрибуты, заданные в CREATE ROLE, могут быть изменены позднее командами ALTER ROLE.

Для добавления и удаления членов ролей, используемых в качестве групп, рекомендуется использовать GRANT и REVOKE.

Предложение VALID UNTIL определяет срок действия только пароля, но не роли как таковой. В частности, ограничение срока пароля не действует при входе пользователя без проверки подлинности по паролю.

Атрибут INHERIT управляет наследованием назначаемых прав (то есть правами доступа к объектам баз данных и членством в ролях). Его действие не распространяется на специальные атрибуты, устанавливаемые командами CREATE ROLE и ALTER ROLE. Например, членства в роли с правом CREATEDB недостаточно для получения права создавать базы данных, даже если установлен атрибут INHERIT; чтобы воспользоваться правом создавать базы данных, необходимо переключиться на эту роль, выполнив SET ROLE.

Атрибут INHERIT устанавливается по умолчанию в целях обратной совместимости: в предыдущих выпусках Postgres Pro пользователи всегда обладали всеми правами групп, в которые они были включены. Однако NOINHERIT по смыслу ближе к тому, что описано в стандарте SQL.

Будьте осторожны с правом CREATEROLE. На роли, создаваемые командой CREATEROLE, не распространяется концепция наследования. Это значит, что даже если роль не имеет определённого права, но может создавать другие роли, она вполне способна создать другую роль с отличным набором прав (за исключением создания ролей с правами суперпользователя). Например, если роль «user» имеет право CREATEROLE, но не CREATEDB, она тем не менее может создать новую роль с правом CREATEDB. Поэтому роль с правом CREATEROLE следует воспринимать как роль почти суперпользователя.

Postgres Pro включает программу createuser, которая предоставляет ту же функциональность, что и команда CREATE ROLE (на самом деле она вызывает эту команду), но может запускаться в командной оболочке.

Ограничение CONNECTION LIMIT действует только приблизительно; если одновременно запускаются два сеанса, тогда как для этой роли остаётся только одно «свободное место», может так случиться, что будут отклонены оба подключения. Кроме того, это ограничение не распространяется на суперпользователей.

Указывая в этой команде незашифрованный пароль, следует проявлять осторожность. Пароль будет передаваться на сервер открытым текстом и может также записываться в историю команд клиента или в протокол работы сервера. Команда createuser, однако, передаёт пароль зашифрованным. Кроме того, в psql есть команда \password, с помощью которой можно безопасно сменить пароль позже.

Примеры

Создание роли, для которой разрешён вход, но не задан пароль:

```
CREATE ROLE jonathan LOGIN;
```

Создание роли с паролем:

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

(CREATE USER действует так же, как CREATE ROLE, но подразумевает ещё и атрибут LOGIN.)

Создание роли с паролем, действующим до конца 2004 г., то есть пароль перестаёт действовать в первую же секунду 2005 г.

```
CREATE ROLE miriam WITH LOGIN PASSWORD 'jw8s0F4' VALID UNTIL '2005-01-01';
```

Создание роли, которая может создавать базы данных и управлять ролями:

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

Совместимость

Оператор CREATE ROLE описан в стандарте SQL, но стандарт требует поддержки только следующего синтаксиса:

```
CREATE ROLE имя [ WITH ADMIN имя_роли ]
```

Возможность создавать множество начальных администраторов и все другие параметры `CREATE ROLE` относятся к расширениям Postgres Pro.

В стандарте SQL определяются концепции пользователей и ролей, но в нём они рассматриваются как отдельные сущности, а все команды создания пользователей считаются внутренней спецификой СУБД. В Postgres Pro мы решили объединить пользователей и роли в единую сущность, так что роли получили дополнительные атрибуты, не описанные в стандарте.

Поведение, наиболее близкое к описанному в стандарте SQL, можно получить, если создавать пользователей с атрибутом `NOINHERIT`, а роли — с атрибутом `INHERIT`.

См. также

SET ROLE, ALTER ROLE, DROP ROLE, GRANT, REVOKE, createuser

CREATE RULE

CREATE RULE — создать правило перезаписи

Синтаксис

```
CREATE [ OR REPLACE ] RULE имя AS ON событие
    TO имя_таблицы [ WHERE условие ]
    DO [ ALSO | INSTEAD ] { NOTHING | команда | ( команда ; команда ... ) }

Здесь допускается событие:

    SELECT | INSERT | UPDATE | DELETE
```

Описание

`CREATE RULE` создаёт правило, применяемое к указанной таблице или представлению. `CREATE OR REPLACE RULE` либо создаёт новое правило, либо заменяет существующее с тем же именем для той же таблицы.

Система правил Postgres Pro позволяет определить альтернативное действие, заменяющее операции добавления, изменения или удаления данных в таблицах базы данных. Грубо говоря, правило описывает дополнительные команды, которые будут выполняться при вызове определённой команды для определённой таблицы. Кроме того, правило `INSTEAD` может заменить заданную команду другой, либо сделать, чтобы она не выполнялась вовсе. Правила также применяются для реализации SQL-запросов. Важно понимать, что правило это фактически механизм преобразования команд (макрос). Заданное преобразование имеет место до начала выполнения команды. Когда требуется выполнить некоторую операцию независимо для каждой физической строки, скорее всего, для этого нужно применять триггер, а не правило. Более подробно о системе правил можно узнать в Главе 38.

В настоящее время правила `ON SELECT` должны быть безусловными, с характеристикой `INSTEAD` (вместо исходного), и их действия должны состоять из единственной команды `SELECT`. Таким образом, правило `ON SELECT` по сути превращает таблицу в представление, чьим видимым содержимым являются строки, возвращаемые командой `SELECT`, заданной в правиле, а не данные, хранящиеся в таблице (если они есть). Вообще же для этой цели лучшим стилем считается пользоваться командой `CREATE VIEW`, а не создавать реальную таблицу и определять затем правило `ON SELECT` для неё.

С помощью правил можно создать иллюзию изменяемого представления, определив правила `ON INSERT`, `ON UPDATE` и `ON DELETE` (либо только те, которых достаточно для решения поставленной задачи) и заменив операции изменения данных в представлении соответствующими действиями с другими таблицами. Если требуется поддерживать оператор `INSERT RETURNING` и подобные ему, в каждое из этих правило обязательно нужно поместить подходящее предложение `RETURNING`.

Использование правил с условиями для сложных изменений представлений связано с одним ограничением: для каждого действия, которое вы хотите разрешить для представления, *необходимо* определить безусловное правило `INSTEAD`. Если определено только условное правило, или правило не типа `INSTEAD`, система отвергнет попытки выполнить изменения, предполагая, что в некоторых случаях изменения могут свестись к операциям с фиктивной нижележащей таблицей. При желании обработать все полезные случаи изменений в условных правилах, добавьте безусловное правило `DO INSTEAD NOTHING`, чтобы система понимала, что ей никогда не придётся изменять нижележащую таблицу. Затем создайте условные правила без свойства `INSTEAD`; в тех случаях, когда они будут применяться, их действия будут добавлены к действию по умолчанию `INSTEAD NOTHING`. (Однако этот способ в настоящее время не подходит для реализации запросов `RETURNING`.)

> **Примечание**
>
> Представления, достаточно простые для реализации автоматического обновления (см. CREATE VIEW), могут быть изменяемыми без пользовательских правил. Хотя вы тем не менее можете создать явное правило, обычно автоматическое преобразование будет работать лучше такого правила.
>
> Другая, заслуживающая рассмотрения, альтернатива правилам — триггеры `INSTEAD OF` (см. CREATE TRIGGER).

Параметры

имя

Имя создаваемого правила. Оно должно отличаться от имён любых других правил для той же таблицы. При наличии нескольких правил для одной таблицы и одного типа события они применяются в алфавитном порядке.

событие

Тип события: `SELECT`, `INSERT`, `UPDATE` или `DELETE`. Заметьте, что команду `INSERT` с предложением `ON CONFLICT` нельзя использовать с таблицами, для которых определены правила `INSERT` или `UPDATE`. В этом случае подумайте о применении изменяемых представлений.

имя_таблицы

Имя (возможно, дополненное схемой) существующей таблицы (или представления), к которой применяется это правило.

условие

Любое выражение условия SQL (возвращающее `boolean`). Это выражение не может ссылаться на какие-либо таблицы, кроме как на `NEW` и `OLD`, и не может содержать агрегатные функции.

`INSTEAD`

`INSTEAD` указывает, что заданные команды должны выполняться *вместо* исходной команды.

`ALSO`

`ALSO` указывает, что заданные команды должны выполняться *в дополнение к* исходной команде.

Если ни `ALSO`, ни `INSTEAD` не указано, по умолчанию подразумевается `ALSO`.

команда

Команда или команды, составляющие действие правила. Здесь допустимы команды: `SELECT`, `INSERT`, `UPDATE`, `DELETE` и `NOTIFY`.

В параметрах *условие* и *команда* можно использовать имена специальных таблиц `NEW` и `OLD` для обращения к значениям в соответствующей таблице. `NEW` (новая) принимается в правилах `ON INSERT` и `ON UPDATE` и обозначает ссылку на новую строку, добавляемую или изменяемую. `OLD` (старая) принимается в правилах `ON UPDATE` и `ON DELETE` и обозначает ссылку на существующую строку, изменяемую или удаляемую.

Замечания

Чтобы создать или изменить правила для таблицы, нужно быть её владельцем.

В правило для `INSERT`, `UPDATE` или `DELETE` для представления можно добавить предложение `RETURNING`, выдающее столбцы представления. Это предложение будет генерировать результат,

если правило сработает при выполнении команды INSERT RETURNING, UPDATE RETURNING или DELETE RETURNING. Когда правило срабатывает при выполнении команды без RETURNING, предложение RETURNING этого правила игнорируется. В текущей реализации только безусловные правила INSTEAD могут содержать RETURNING; более того, допускается максимум одно предложение RETURNING среди всех правил для некоторого события. (Благодаря этому ограничению, только одно предложение RETURNING может быть выбрано для вычисления результатов.) Запросы с RETURNING к данному представлению не будут выполняться, если ни одно из определённых для него правил не содержит предложение RETURNING.

Очень важно следить за тем, чтобы правила не зацикливались. Например, два следующих определения правил будут приняты Postgres Pro, но при попытке выполнить команду SELECT Postgres Pro сообщит об ошибке из-за рекурсивного расширения правила:

```
CREATE RULE "_RETURN" AS
    ON SELECT TO t1
    DO INSTEAD
        SELECT * FROM t2;

CREATE RULE "_RETURN" AS
    ON SELECT TO t2
    DO INSTEAD
        SELECT * FROM t1;

SELECT * FROM t1;
```

В настоящее время, если действие правила содержит команду NOTIFY, эта команда будет выполняться безусловно, то есть, NOTIFY будет выдаваться, даже если не найдётся никаких строк, к которым бы применялось правило. Например, в следующем примере:

```
CREATE RULE notify_me AS ON UPDATE TO mytable DO ALSO NOTIFY mytable;

UPDATE mytable SET name = 'foo' WHERE id = 42;
```

одно событие NOTIFY будет отправлено при выполнении команды UPDATE, даже если никакие строки не соответствуют условию id = 42. Это недостаток текущей реализации, который может быть исправлен в будущих версиях.

Совместимость

Оператор CREATE RULE является языковым расширением Postgres Pro, как и вся система перезаписи запросов.

См. также

ALTER RULE, DROP RULE

CREATE SCHEMA

CREATE SCHEMA — создать схему

Синтаксис

```
CREATE SCHEMA имя_схемы [ AUTHORIZATION указание_роли ] [ элемент_схемы [ ... ] ]
CREATE SCHEMA AUTHORIZATION указание_роли [ элемент_схемы [ ... ] ]
CREATE SCHEMA IF NOT EXISTS имя_схемы [ AUTHORIZATION указание_роли ]
CREATE SCHEMA IF NOT EXISTS AUTHORIZATION указание_роли

Здесь указание_роли:

    имя_пользователя
  | CURRENT_USER
  | SESSION_USER
```

Описание

`CREATE SCHEMA` создаёт новую схему в текущей базе данных. Имя схемы должно отличаться от имён других существующих схем в текущей базе данных.

Схема по сути представляет собой пространство имён: она содержит именованные объекты (таблицы, типы данных, функции и операторы), имена которых могут совпадать с именами других объектов, существующих в других схемах. Для обращения к объекту нужно либо «дополнить» его имя именем схемы в виде префикса, либо установить путь поиска, включающий требуемую схему. Команда `CREATE`, в которой указывается неполное имя объекта, создаёт объект в текущей схеме (схеме, стоящей первой в пути поиска; узнать её позволяет функция `current_schema`).

Команда `CREATE SCHEMA` может дополнительно включать подкоманды, создающие объекты в новой схеме. Эти подкоманды по сути воспринимаются как отдельные команды, выполняемые после создания схемы, за исключением того, что с предложением `AUTHORIZATION` все создаваемые объекты будут принадлежать указанному в нём пользователю.

Параметры

имя_схемы

Имя создаваемой схемы. Если оно опущено, именем схемы будет *имя_пользователя*. Это имя не может начинаться с `pg_`, так как такие имена зарезервированы для системных схем.

имя_пользователя

Имя пользователя (роли), назначаемого владельцем новой схемы. Если опущено, по умолчанию владельцем будет пользователь, выполняющий команды. Чтобы назначить владельцем создаваемой схемы другую роль, необходимо быть непосредственным или опосредованным членом этой роли, либо суперпользователем.

элемент_схемы

Оператор SQL, определяющий объект, создаваемый в новой схеме. В настоящее время `CREATE SCHEMA` может содержать только подкоманды `CREATE TABLE`, `CREATE VIEW`, `CREATE INDEX`, `CREATE SEQUENCE`, `CREATE TRIGGER` и `GRANT`. Создать объекты других типов можно отдельными командами после создания схемы.

`IF NOT EXISTS`

Не делать ничего (только выдать замечание), если схема с таким именем уже существует. Когда используется это указание, эта команда не может содержать подкоманды *элемент_схемы*.

Замечания

Чтобы создать схему, пользователь должен иметь право `CREATE` в текущей базе данных. (Разумеется, на суперпользователей это условие не распространяется.)

Примеры

Создание схемы:

```
CREATE SCHEMA myschema;
```

Создание схемы для пользователя `joe`; схема так же получит имя `joe`:

```
CREATE SCHEMA AUTHORIZATION joe;
```

Создание схемы с именем `test`, владельцем которой будет пользователь `joe`, если только схема `test` ещё не существует. (Является ли владельцем существующей схемы пользователь `joe`, значения не имеет.)

```
CREATE SCHEMA IF NOT EXISTS test AUTHORIZATION joe;
```

Создание схемы, в которой сразу создаются таблица и представление:

```
CREATE SCHEMA hollywood
    CREATE TABLE films (title text, release date, awards text[])
    CREATE VIEW winners AS
        SELECT title, release FROM films WHERE awards IS NOT NULL;
```

Заметьте, что отдельные подкоманды не завершаются точкой с запятой.

Следующие команды приводят к тому же результату другим способом:

```
CREATE SCHEMA hollywood;
CREATE TABLE hollywood.films (title text, release date, awards text[]);
CREATE VIEW hollywood.winners AS
    SELECT title, release FROM hollywood.films WHERE awards IS NOT NULL;
```

Совместимость

Стандарт SQL также допускает в команде `CREATE SCHEMA` предложение `DEFAULT CHARACTER SET` и дополнительные типы подкоманд, которые Postgres Pro в настоящее время не принимает.

В стандарте SQL говорится, что подкоманды в `CREATE SCHEMA` могут следовать в любом порядке. Однако текущая реализация в Postgres Pro не воспринимает все возможные варианты ссылок вперёд в подкомандах, поэтому иногда возникает необходимость переупорядочить подкоманды, чтобы исключить такие ссылки.

Согласно стандарту SQL, владелец схемы всегда владеет всеми объектами в ней, но Postgres Pro допускает размещение в схемах объектов, принадлежащих не владельцу схемы. Такая ситуация возможна, только если владелец схемы даст право `CREATE` в этой схеме кому-либо другому, либо объекты в ней будет создавать суперпользователь.

Указание `IF NOT EXISTS` является расширением Postgres Pro.

См. также

ALTER SCHEMA, DROP SCHEMA

CREATE SEQUENCE

CREATE SEQUENCE — создать генератор последовательности

Синтаксис

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE [ IF NOT EXISTS ] имя
    [ AS тип_данных ]
    [ INCREMENT [ BY ] шаг ]
    [ MINVALUE мин_значение | NO MINVALUE ] [ MAXVALUE макс_значение | NO MAXVALUE ]
    [ START [ WITH ] начало ] [ CACHE кеш ] [ [ NO ] CYCLE ]
    [ OWNED BY { имя_таблицы.имя_столбца | NONE } ]
```

Описание

`CREATE SEQUENCE` создаёт генератор последовательности. Эта операция включает создание и инициализацию специальной таблицы *имя*, содержащей одну строку. Владельцем генератора будет пользователь, выполняющий эту команду.

Если указано имя схемы, последовательность создаётся в заданной схеме, в противном случае — в текущей. Временные последовательности существуют в специальной схеме, так что при создании таких последовательностей имя схемы задать нельзя. Имя последовательности должно отличаться от имён других последовательностей, таблиц, индексов, представлений или сторонних таблиц, уже существующих в этой схеме.

После создания последовательности работать с ней можно, вызывая функции `nextval`, `currval` и `setval`. Эти функции документированы в Разделе 9.16.

Хотя непосредственно изменить значение последовательности нельзя, получить её параметры и текущее состояние можно таким запросом:

```
SELECT * FROM name;
```

В частности, поле `last_value` последовательности будет содержать последнее значение, выделенное для какого-либо сеанса. (Конечно, ко времени вывода это значение может стать неактуальным, если другие сеансы активно вызывают `nextval`.)

Параметры

`TEMPORARY` или `TEMP`

Если указано, объект последовательности создаётся только для данного сеанса и автоматически удаляется при завершении сеанса. Существующая постоянная последовательность с тем же именем не будут видна (в этом сеансе), пока существует временная, однако к ней можно обратиться, дополнив имя указанием схемы.

`IF NOT EXISTS`

Не считать ошибкой, если отношение с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее отношение как-то соотносится с последовательностью, которая могла бы быть создана — это может быть даже не последовательность.

имя

Имя создаваемой последовательности (возможно, дополненное схемой).

тип_данных

Необязательное предложение `AS` *тип_данных* задаёт тип данных для последовательности. Допустимые типы: `smallint`, `integer` и `bigint`. По умолчанию устанавливается тип `bigint`.

От типа данных зависят принимаемые по умолчанию минимальное и максимальное значения последовательности.

шаг

Необязательное предложение `INCREMENT BY` *шаг* определяет, какое число будет добавляться к текущему значению последовательности для получения нового значения. С положительным шагом последовательность будет возрастающей, а с отрицательным — убывающей. Значение по умолчанию: 1.

мин_значение
`NO MINVALUE`

Необязательное предложение `MINVALUE` *мин_значение* определяет наименьшее число, которое будет генерировать последовательность. Если это предложение опущено либо указано `NO MINVALUE`, используется значение по умолчанию: 1 для возрастающей последовательности или минимальное значение типа данных — для убывающей.

макс_значение
`NO MAXVALUE`

Необязательное предложение `MAXVALUE` *макс_значение* определяет наибольшее число, которое будет генерировать последовательность. Если это предложение опущено либо указано `NO MAXVALUE`, используется значение по умолчанию: максимальное значение типа данных для возрастающей последовательности или -1 — для убывающей.

начало

Необязательное предложение `START WITH` *начало* позволяет запустить последовательность с любого значения. По умолчанию началом считается *мин_значение* для возрастающих последовательностей и *макс_значение* для убывающих.

кеш

Необязательное предложение `CACHE` *кеш* определяет, сколько чисел последовательности будет выделяться и сохраняться в памяти для ускорения доступа к ним. Минимальное значение равно 1 (за один раз генерируется только одно значение, т. е. кеширования нет), и оно же предполагается по умолчанию.

`CYCLE`
`NO CYCLE`

Параметр `CYCLE` позволяет зациклить последовательность при достижении *макс_значения* или *мин_значения* для возрастающей или убывающей последовательности, соответственно. Когда этот предел достигается, следующим числом этих последовательностей будет соответственно *мин_значение* или *макс_значение*.

Если указывается `NO CYCLE`, при каждом вызове `nextval` после достижения предельного значения будет возникать ошибка. Если указания `CYCLE` и `NO CYCLE` отсутствуют, по умолчанию предполагается `NO CYCLE`.

`OWNED BY` *имя_таблицы.имя_столбца*
`OWNED BY NONE`

Предложение `OWNED BY` позволяет связать последовательность с определённым столбцом таблицы так, чтобы при удалении этого столбца (или всей таблицы) последовательность удалялась автоматически. Указанная таблица должна иметь того же владельца и находиться в той же схеме, что и последовательность. Подразумеваемое по умолчанию предложение `OWNED BY NONE` указывает, что такая связь не устанавливается.

Замечания

Для удаления последовательности применяется команда `DROP SEQUENCE`.

Последовательности основаны на арифметике bigint, так что их значения не могут выходить за диапазон восьмибайтовых целых (-9223372036854775808 .. 9223372036854775807).

Так как вызовы nextval и setval никогда не откатываются, объекты последовательностей не подходят, если требуется обеспечить непрерывное назначение номеров последовательностей. Непрерывное назначение можно организовать, используя исключительную блокировку таблицы со счётчиком; однако это решение будет гораздо дороже, чем применение объектов последовательностей, особенно когда последовательные номера будут затребоваться сразу многими транзакциями.

Если значение параметра *кеш* больше единицы, и объект последовательности используется параллельно в нескольких сеансах, результат может оказаться не вполне ожидаемым. Каждый сеанс будет выделять и кешировать несколько очередных значений последовательности при одном обращении к объекту последовательности и соответственно увеличивать последнее_значение этого объекта. Затем при следующих *кеш*-1 вызовах nextval в этом сеансе будет просто возвращать заготовленные значения, не касаясь объекта последовательности. В результате, все числа, выделенные, но не использованные в сеансе, будут потеряны при завершении сеанса, что приведёт к образовании «дырок» в последовательности.

Более того, хотя разным сеансам гарантированно выделяются различные значения последовательности, если рассмотреть все сеансы в целом, порядок этих значений может быть нарушен. Например, при значении *кеш*, равном 10, сеанс A может зарезервировать значения 1..10 и получить nextval=1, затем сеанс B может зарезервировать значения 11..20 и получить nextval=11 до того, как в сеансе A сгенерируется nextval=2. Таким образом, при значении *кеш*, равном одному, можно быть уверенными в том, что nextval генерирует последовательные значения; но если *кеш* больше одного, рассчитывать можно только на то, что все значения nextval различны; их порядок может быть непоследовательным. Кроме того, last_value возвращает последнее зарезервированное значение для всех сеансов, вне зависимости от того, было ли оно уже возвращено функцией nextval.

Также следует учитывать, что действие функции setval, выполненной для такой последовательности, проявится в других сеансах только после того, как в них будут использованы все предварительно закешированные значения.

Примеры

Создание возрастающей последовательности с именем serial, с начальным значением 101:

```
CREATE SEQUENCE serial START 101;
```

Получение следующего номера этой последовательности:

```
SELECT nextval('serial');

 nextval
---------
     101
```

Получение следующего номера этой последовательности:

```
SELECT nextval('serial');

 nextval
---------
     102
```

Использование этой последовательности в команде INSERT:

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

Изменение значения последовательности после COPY FROM:

```
BEGIN;
COPY distributors FROM 'input_file';
SELECT setval('serial', max(id)) FROM distributors;
END;
```

Совместимость

Команда `CREATE SEQUENCE` соответствует стандарту SQL, со следующими исключениями:

- Для получения следующего значения применяется функция `nextval()`, а не выражение `NEXT VALUE FOR`, как того требует стандарт.
- Предложение `OWNED BY` является расширением Postgres Pro.

См. также

ALTER SEQUENCE, DROP SEQUENCE

CREATE SERVER

CREATE SERVER — создать сторонний сервер

Синтаксис

```
CREATE SERVER [IF NOT EXISTS] имя_сервера [ TYPE 'тип_сервера' ] [ VERSION
 'версия_сервера' ]
    FOREIGN DATA WRAPPER имя_обёртки_сторонних_данных
    [ OPTIONS ( параметр 'значение' [, ... ] ) ]
```

Описание

`CREATE SERVER` создаёт сторонний сервер. Владельцем сервера становится создавший его пользователь.

Определение стороннего сервера обычно включает информацию о подключении, которую использует обёртка сторонних данных для доступа к внешнему ресурсу. Определяя сопоставления пользователей, можно установить и другие параметры подключения, связанные с пользователями.

Имя сервера должно быть уникальным в базе данных.

Для создания сервера требуется право `USAGE` для обёртки сторонних данных.

Параметры

`IF NOT EXISTS`

Не считать ошибкой, если сервер с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующий сервер как-то соотносится с тем, который мог бы быть создан.

имя_сервера

Имя создаваемого стороннего сервера.

тип_сервера

Необязательный тип сервера, может быть полезен для обёрток сторонних данных.

версия_сервера

Необязательная версия сервера, может быть полезна для обёрток сторонних данных.

имя_обёртки_сторонних_данных

Имя обёртки сторонних данных, управляющей сервером.

`OPTIONS ( параметр 'значение' [, ... ] )`

Это предложение определяет параметры сервера. Эти параметры обычно задают свойства подключения к серверу; их конкретные имена и значения зависят от обёртки сторонних данных.

Замечания

При использовании модуля dblink имя стороннего сервера может служить аргументом функции dblink_connect, определяющим параметры подключения. Для такого варианта использования необходимо иметь право `USAGE` для стороннего сервера.

Примеры

Создание сервера `myserver`, доступного через обёртку `postgres_fdw`:

```
CREATE SERVER myserver FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'foo', dbname
 'foodb', port '5432');
```

За подробностями обратитесь к postgres_fdw.

Совместимость

`CREATE SERVER` соответствует стандарту ISO/IEC 9075-9 (SQL/MED).

См. также

ALTER SERVER, DROP SERVER, CREATE FOREIGN DATA WRAPPER, CREATE FOREIGN TABLE, CREATE USER MAPPING

CREATE STATISTICS

CREATE STATISTICS — создать расширенную статистику

Синтаксис

```
CREATE STATISTICS [ IF NOT EXISTS ] имя_статистики
    [ ( вид_статистики [, ... ] ) ]
    ON имя_столбца, имя_столбца [, ...]
    FROM имя_таблицы
```

Описание

Команда `CREATE STATISTICS` создаст новый объект расширенной статистики, отслеживающий данные определённой таблицы, сторонней таблицы или материализованного представления. Объект статистики будет создан в текущей базе данных, и его владельцем станет пользователь, выполняющий команду.

Если задано имя схемы (например, `CREATE STATISTICS myschema.mystat ...`), объект статистики создаётся в указанной схеме, в противном случае — в текущей. Имя объекта статистики должно отличаться от имён других объектов статистики в этой схеме.

Параметры

`IF NOT EXISTS`

Не считать ошибкой, если объект статистики с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что при этом проверяется только имя объекта, а не характеристики его определения.

имя_статистики

Имя создаваемого объекта статистики (возможно, дополненное схемой).

вид_статистики

Вид статистики, которая будет вычисляться в этом объекте. В настоящее время поддерживается статистика `ndistinct`, подсчёт числа различных значений, и `dependencies`, определение функциональных зависимостей. Если это предложение опущено, в объект статистики включаются все поддерживаемые виды статистики. За дополнительными сведениями обратитесь к Подразделу 14.2.2 и Разделу 65.2.

имя_столбца

Имя столбца таблицы, который будет покрываться вычисляемой статистикой. Необходимо указать имена минимум двух столбцов.

имя_таблицы

Имя (возможно, дополненное схемой) таблицы, содержащей столбцы, по которым создаётся статистика; об особенностях, связанных с наследованием и секционированием, рассказывается в описании ANALYZE.

Замечания

Чтобы создать объект статистики, читающий таблицу, необходимо быть владельцем этой таблицы. После создания объекта статистики его владелец может определяться независимо от нижележащей таблицы.

Примеры

Создайте таблицу `t1` с двумя функционально зависимыми столбцами; то есть знания значения одного столбца достаточно, чтобы определить значение другого. Затем для этих столбцов постройте статистику функциональной зависимости:

```
CREATE TABLE t1 (
    a   int,
    b   int
);

INSERT INTO t1 SELECT i/100, i/500
                FROM generate_series(1,1000000) s(i);

ANALYZE t1;

-- число совпадающих строк будет катастрофически недооценено:
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);

CREATE STATISTICS s1 (dependencies) ON a, b FROM t1;

ANALYZE t1;

-- теперь оценка числа строк стала точнее:
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);
```

Без статистики функциональной зависимости планировщик предположил бы, что два условия `WHERE` независимы друг от друга, и перемножил бы их оценки избирательности, что дало бы слишком заниженную оценку числа строк. Однако с созданной статистикой планировщик понимает, что условия `WHERE` избыточны, и не ошибается с этой оценкой.

Совместимость

Команда `CREATE STATISTICS` отсутствует в стандарте SQL.

См. также

ALTER STATISTICS, DROP STATISTICS

CREATE SUBSCRIPTION

CREATE SUBSCRIPTION — создать подписку

Синтаксис

```
CREATE SUBSCRIPTION имя_подписки
    CONNECTION 'строка_подключения'
    PUBLICATION имя_публикации [, ...]
    [ WITH ( параметр_подписки [= значение] [, ... ] ) ]
```

Описание

`CREATE SUBSCRIPTION` создаёт подписку для текущей базы данных. Имя подписки должно отличаться от имён других существующих подписок в текущей базе.

Подписка представляет собой реплицирующее подключение к публикующему серверу. Поэтому данная команда не только добавляет определения подписки в локальные каталоги, но также создаёт слот репликации на удалённом сервере.

В момент фиксации транзакции, в рамках которой выполняется эта команда, будет запущен рабочий процесс логической репликации.

Дополнительные сведения о подписках и логической репликации в целом можно найти в Разделе 30.2 и Главе 30.

Параметры

имя_подписки

Имя новой подписки.

`CONNECTION '`*строка_подключения*`'`

Строка подключения к публикующему серверу. Подробности описаны в Подразделе 31.1.1.

`PUBLICATION` *имя_публикации*

Имена публикаций на публикующем сервере, на которые оформляется подписка.

`WITH (` *параметр_подписки* `[= `*значение*`] [, ... ] )`

В этом предложении могут задаваться следующие необязательные параметры подписки:

`copy_data` (`boolean`)

Определяет, должны ли копироваться существующие данные в публикациях, на которые оформляется подписка, сразу после начала репликации. Значение по умолчанию — `true`.

`create_slot` (`boolean`)

Определяет, должна ли команда создавать слот репликации на публикующем сервере. Значение по умолчанию — `true`.

`enabled` (`boolean`)

Определяет, активировать ли репликацию в подписке, или её нужно только настроить, но не запускать сразу. Значение по умолчанию — `true`.

`slot_name` (`string`)

Имя слота репликации, которое должно использоваться. По умолчанию в качестве имени слота используется имя подписки.

Когда в качестве slot_name задаётся NONE, с подпиской не будет связан слот репликации. Это может быть полезно, если слот репликации позднее будет создаваться вручную. У таких подписок также должны быть равны false свойства enabled и create_slot.

synchronous_commit (enum)

Значение этого параметра переопределяет свойство synchronous_commit. По умолчанию — off.

Значение off безопасно для логической репликации: если подписчик потеряет транзакции из-за нарушения синхронизации, данные будут повторно переданы с публикующего сервера.

При выполнении синхронной логической репликации может быть уместно другое значение. Рабочие процессы логической репликации передают позиции записанных и сохранённых на диске данных публикующему серверу, так что при синхронной репликации он будет ждать завершения сохранения. Это значит, что значение off параметра synchronous_commit на подписчике может увеличить задержку при выполнении COMMIT на сервере публикации. При таком сценарии может быть выгоднее задать для synchronous_commit значение local или выше.

connect (boolean)

Определяет, нужно ли при выполнении CREATE SUBSCRIPTION подключаться к публикующему серверу. Если равняется false, значениями по умолчанию параметров enabled, create_slot и copy_data тоже будет false.

Значение false параметра connect несовместимо со значением true параметров enabled, create_slot и copy_data.

Так как со значением false соединение не устанавливается, подписка на таблицы не оформляется, так что после включения подписки ничего не будет реплицироваться. Чтобы таблицы вошли в подписку, потребуется позже выполнить ALTER SUBSCRIPTION ... REFRESH PUBLICATION.

Замечания

Подробнее о том, как организовать управление доступом подписчиков к публикующему серверу, рассказывается в Разделе 30.7.

При создании слота репликации (поведение по умолчанию) CREATE SUBSCRIPTION нельзя выполнять внутри блока транзакции.

Создание подписки с подключением к тому же кластеру баз данных (например, для организации репликации между базами данных в одном кластере или в одной базе данных) будет успешным, только если слот репликации не создаётся той же командой. В противном случае команда CREATE SUBSCRIPTION зависнет. Чтобы оформить такую подписку, слот репликации нужно создать отдельно (воспользовавшись функцией pg_create_logical_replication_slot и передав ей имя модуля pgoutput) и создать подписку с параметром create_slot = false. Это ограничение реализации, которое может быть устранено в будущем выпуске.

Примеры

Создание подписки на репликации mypublication и insert_only на удалённом сервере с немедленным запуском репликации при фиксировании транзакции:

```
CREATE SUBSCRIPTION mysub
         CONNECTION 'host=192.168.1.50 port=5432 user=foo dbname=foodb'
        PUBLICATION mypublication, insert_only;
```

Создание подписки на публикацию insert_only на удалённом сервере с отключением репликации для запуска в будущем.

```
CREATE SUBSCRIPTION mysub
         CONNECTION 'host=192.168.1.50 port=5432 user=foo dbname=foodb'
        PUBLICATION insert_only
               WITH (enabled = false);
```

Совместимость

`CREATE SUBSCRIPTION` является расширением Postgres Pro.

См. также

ALTER SUBSCRIPTION, DROP SUBSCRIPTION, CREATE PUBLICATION, ALTER PUBLICATION

CREATE TABLE

CREATE TABLE — создать таблицу

Синтаксис

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
 EXISTS ] имя_таблицы ( [
  { имя_столбца тип_данных [ COLLATE правило_сортировки ] [ ограничение_столбца
 [ ... ] ]
    | ограничение_таблицы
    | LIKE исходная_таблица [ вариант_копирования ... ] }
    [, ... ]
] )
[ INHERITS ( таблица_родитель [, ... ] ) ]
[ PARTITION BY { RANGE | LIST } ( { имя_столбца | ( выражение ) }
 [ COLLATE правило_сортировки ] [ класс_операторов ] [, ... ] ) ]
[ WITH ( параметр_хранения [= значение] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE табл_пространство ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
 EXISTS ] имя_таблицы
    OF имя_типа [ (
  { имя_столбца [ WITH OPTIONS ] [ ограничение_столбца [ ... ] ]
    | ограничение_таблицы }
    [, ... ]
) ]
[ PARTITION BY { RANGE | LIST } ( { имя_столбца | ( выражение ) }
 [ COLLATE правило_сортировки ] [ класс_операторов ] [, ... ] ) ]
[ WITH ( параметр_хранения [= значение] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE табл_пространство ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
 EXISTS ] имя_таблицы
    PARTITION OF таблица_родитель [ (
  { имя_столбца [ WITH OPTIONS ] [ ограничение_столбца [ ... ] ]
    | ограничение_таблицы }
    [, ... ]
) ] FOR VALUES указание_границ_секции
[ PARTITION BY { RANGE | LIST } ( { имя_столбца | ( выражение ) }
 [ COLLATE правило_сортировки ] [ класс_операторов ] [, ... ] ) ]
[ WITH ( параметр_хранения [= значение] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE табл_пространство ]

Здесь ограничение_столбца:

[ CONSTRAINT имя_ограничения ]
{ NOT NULL |
  NULL |
  CHECK ( выражение ) [ NO INHERIT ] |
  DEFAULT выражение_по_умолчанию |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( параметры_последовательности ) ] |
  UNIQUE параметры_индекса |
```

```
  PRIMARY KEY параметры_индекса |
  REFERENCES целевая_таблица [ ( целевой_столбец ) ] [ MATCH FULL | MATCH PARTIAL |
 MATCH SIMPLE ]
    [ ON DELETE действие ] [ ON UPDATE действие ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

и ограничение_таблицы:

[ CONSTRAINT имя_ограничения ]
{ CHECK ( выражение ) [ NO INHERIT ] |
  UNIQUE ( имя_столбца [, ... ] ) параметры_индекса |
  PRIMARY KEY ( имя_столбца [, ... ] ) параметры_индекса |
  EXCLUDE [ USING индексный_метод ] ( элемент_исключения WITH оператор
 [, ... ] ) параметры_индекса [ WHERE ( предикат ) ] |
  FOREIGN KEY ( имя_столбца [, ... ] ) REFERENCES целевая_таблица [ ( целевой_столбец
 [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE действие ] [ ON
 UPDATE действие ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

и вариант_копирования:

{ INCLUDING | EXCLUDING } { COMMENTS | CONSTRAINTS | DEFAULTS | IDENTITY | INDEXES |
 STATISTICS | STORAGE | ALL }

и указание_границ_секции:

IN ( { числовая_константа | строковая_константа | TRUE | FALSE | NULL } [, ...] ) |
FROM ( { числовая_константа | строковая_константа | TRUE | FALSE | MINVALUE |
 MAXVALUE } [, ...] )
  TO ( { числовая_константа | строковая_константа | TRUE | FALSE | MINVALUE |
 MAXVALUE } [, ...] )

параметры_индекса в ограничениях UNIQUE, PRIMARY KEY и EXCLUDE:

[ INCLUDE ( имя_столбца [, ... ] ) ]
[ WITH ( параметр_хранения [= значение] [, ... ] ) ]
[ USING INDEX TABLESPACE табл_пространство ]

элемент_исключения в ограничении EXCLUDE:

{ имя_столбца | ( выражение ) } [ класс_операторов ] [ ASC | DESC ] [ NULLS { FIRST |
 LAST } ]
```

Описание

`CREATE TABLE` создаёт новую, изначально пустую таблицу в текущей базе данных. Владельцем таблицы будет пользователь, выполнивший эту команду.

Если задано имя схемы (например, `CREATE TABLE myschema.mytable ...`), таблица создаётся в указанной схеме, в противном случае — в текущей. Временные таблицы существуют в специальной схеме, так что при создании таких таблиц имя схемы задать нельзя. Имя таблицы должно отличаться от имён других таблиц, последовательностей, индексов, представлений или сторонних таблиц в этой схеме.

`CREATE TABLE` также автоматически создаёт составной тип данных, соответствующий одной строке таблицы. Таким образом, имя таблицы не может совпадать с именем существующего типа в этой же схеме.

Необязательные предложения ограничений задают ограничения (проверки), которым должны удовлетворять добавляемые или изменяемые строки, чтобы операция добавления или изменения была выполнена успешно. Ограничение представляет собой SQL-объект, помогающий некоторым способом определить множество допустимых значений в таблице.

Определить ограничения можно двумя способами: в виде ограничений таблицы и в виде ограничений столбца. Ограничение столбца определяется как часть определения столбца, а ограничение таблицы не привязывается к конкретному столбцу и может задействовать несколько столбцов. Любые ограничения столбцов можно также записать в виде ограничения таблицы, они введены просто для удобства записи в случаях, когда ограничение затрагивает только один столбец.

Чтобы создать таблицу, необходимо иметь право `USAGE` для типов всех столбцов или типа в предложении `OF`, соответственно.

Параметры

`TEMPORARY` или `TEMP`

С таким указанием таблица создаётся как временная. Временные таблицы автоматически удаляются в конце сеанса или могут удаляться в конце текущей транзакции (см. описание `ON COMMIT` ниже). Существующая постоянная таблица с тем же именем не будет видна в текущем сеансе, пока существует временная, однако к ней можно обратиться, дополнив имя указанием схемы. Все индексы, создаваемые для временной таблицы, так же автоматически становятся временными.

Демон автоочистки не может прочитать и, как следствие, сжимать и анализировать временные таблицы. По этой причине соответствующие операции очистки и анализа следует выполнять, вызывая SQL-команды в рамках сеанса. Например, если временную таблицу планируется использовать в сложных запросах, будет разумным выполнить для неё `ANALYZE` после того, как она будет наполнена.

По желанию можно добавить указание `GLOBAL` или `LOCAL` перед `TEMPORARY` или `TEMP`. В настоящее время это не имеет значения для Postgres Pro и считается устаревшей возможностью; см. Раздел «Совместимость».

`UNLOGGED`

С этим указанием таблица создаётся как нежурналируемая. Данные, записываемые в нежурналируемые таблицы, не проходят через журнал предзаписи (см. Главу 29), в результате чего такие таблицы работают гораздо быстрее обычных. Однако они не защищены от сбоя; при сбое или аварийном отключении сервера нежурналируемая таблица автоматически усекается. Кроме того, содержимое нежурналируемой таблицы не реплицируется на ведомые серверы. Любые индексы, создаваемые для нежурналируемой таблицы, автоматически становятся нежурналируемыми.

`IF NOT EXISTS`

Не считать ошибкой, если отношение с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее отношение как-то соотносится с тем, которое могло бы быть создано.

`имя_таблицы`

Имя создаваемой таблицы (возможно, дополненное схемой).

`OF` *`имя_типа`*

Создаёт *типизированную таблицу*, структура которой определяется указанным составным типом (его имя может быть дополнено схемой). Типизированная таблица привязана к породившему её типу; например, при удалении типа (командой `DROP TYPE ... CASCADE`) будет удалена и эта таблица.

Когда создаётся типизированная таблица, типы данных её столбцов определяются нижележащим составным типом, а не задаются командой CREATE TABLE. Но CREATE TABLE может добавить в таблицу значения по умолчанию и ограничения, а также задать параметры её хранения.

PARTITION OF *таблица_родитель* FOR VALUES *указание_границ_секции*

Создаёт таблицу как *секцию* указанной родительской таблицы.

Здесь *указание_границ_секции* должно соответствовать методу и ключу секционирования родительской таблицы и не должно конфликтовать с любой существующей секцией того же родителя. Вариант указания с IN используется для секционирования по спискам, а вариант с FROM и TO для секционирования по диапазонам.

В качестве значений, задаваемых в *указании_границ_секции*, принимается буквальное значение, NULL и MINVALUE или MAXVALUE. Буквальным значением должна быть либо числовая константа, приводимая к типу соответствующего столбца ключа разбиения, либо строковая константа, представляющая допустимое входное значение для этого типа.

При создании секции с разбиением по спискам возможно указать, что столбец ключа разбиения может содержать NULL, включив в список секции NULL. Однако у отдельно взятой родительской таблицы может быть не больше одной такой секции. Для диапазонных секций NULL задать нельзя.

При создании диапазонной секции нижняя граница, задаваемая во FROM, включается в диапазон, а верхняя граница, задаваемая в TO — исключается. То есть, значения, задаваемые в списке FROM, являются допустимыми значениями соответствующих столбцов ключа разбиения для этой секции, а значения в списке TO — нет. Заметьте, что это утверждение должно восприниматься с учётом правил сравнения строк таблицы (см. Подраздел 9.23.5). Например, с разбиением PARTITION BY RANGE (x,y), секция с границами FROM (1, 2) TO (3, 4) примет x=1 с любым значением y>=2, x=2 с любым y, отличным от NULL, и x=3 с любым y<4.

Специальные значения MINVALUE и MAXVALUE могут использоваться при создании диапазонной секции для указания, что нижняя или верхняя граница для значений столбца отсутствует. Например, секция, определённая с указанием FROM (MINVALUE) TO (10), будет принимать любые значения меньше 10, а секция, определённая с указанием FROM (10) TO (MAXVALUE), — любые значения, которые больше или равны 10.

При создании диапазонной секции с более чем одним столбцом может также иметь смысл использовать MAXVALUE в определении нижней границы, а MINVALUE — верхней. Например, секция, определённая с указанием FROM (0, MAXVALUE) TO (10, MAXVALUE), будет принимать любые строки, в которых первый столбец ключа разбиения больше 0 и меньше или равен 10. Подобным образом, секция, определённая с указанием FROM ('a', MINVALUE) TO ('b', MINVALUE), будет принимать строки, в которых первый столбец ключа разбиения начинается с «a».

Заметьте, что если для одного столбца в границе секции задаётся MINVALUE или MAXVALUE, то же значение должно применяться и для всех последующих столбцов. Например, граница (10, MINVALUE, 0) будет некорректной; допустимая граница: (10, MINVALUE, MINVALUE).

Также заметьте, что для некоторых типов элементов, таких как timestamp, наряду с другими значениями допускается значение "infinity" (бесконечность). Оно отличается от вариантов MINVALUE и MAXVALUE, которые на самом деле не обозначают значения, которые можно сохранить, а просто говорят о том, это значение не ограничено. MAXVALUE можно воспринимать как значение, которое больше любого другого, включая "бесконечность", а MINVALUE меньше любого другого значения, включая "минус бесконечность". Таким образом, диапазон FROM ('infinity') TO (MAXVALUE) не будет пустым, а будет принимать ровно одно значение — «infinity».

В секции должны содержаться столбцы с теми же именами и типами, что и в секционированной таблице, к которой она относится. Если родитель имеет характеристику `WITH OIDS`, все секции должны иметь OID; родительский столбец OID будет наследоваться всеми секциями, как и любой другой столбец. Изменение имён и типов столбцов в секционируемой таблице, а также добавление или удаление столбца OID будет автоматически распространяться во все секции. Ограничения `CHECK` будут наследоваться автоматически всеми секциями, но для отдельных секций могут быть заданы дополнительные ограничения `CHECK`; дополнительные ограничения с теми же именами и условиями, как в родительской таблице, будут объединены с родительским ограничением. Также независимо для каждой секции могут быть заданы значения по умолчанию. Но заметьте, что значение по умолчанию, заданное на уровне секции, не будет действовать при добавлении строк через секционированную таблицу.

Строки, добавляемые в секционированную таблицу, будут автоматически перенаправляться в соответствующую секцию. Если подходящей секции не найдётся, произойдёт ошибка. Также ошибка произойдёт, если при изменении строки в определённой секции требуется переместить её в другую секцию из-за изменения значений в ключе разбиения.

Такие операции, как TRUNCATE, обычно затрагивают и саму таблицу, и каскадно распространяются на все дочерние секции, но могут также выполняться в отдельных секциях. Заметьте, что для удаления секции с помощью `DROP TABLE` требуется установить блокировку `ACCESS EXCLUSIVE` в родительской таблице.

имя_столбца

Имя столбца, создаваемого в новой таблице.

тип_данных

Тип данных столбца (может включать определение массива с этим типом). За дополнительными сведениями о типах данных, которые поддерживает Postgres Pro, обратитесь к Главе 8.

`COLLATE` *правило_сортировки*

Предложение `COLLATE` назначает правило сортировки для столбца (который должен иметь тип, поддерживающий сортировку). Если оно отсутствует, используется правило сортировки по умолчанию, установленное для типа данных столбца.

`INHERITS ( таблица_родитель [, ... ] )`

Необязательное предложение `INHERITS` определяет список таблиц, от которых новая таблица будет автоматически наследовать все столбцы. Родительские таблицы могут быть обычными или сторонними таблицами.

При использовании `INHERITS` создаётся постоянная связь дочерней таблицы с родительскими. Изменения схемы в родительских таблицах обычно также отражаются в дочерних, и по умолчанию при чтении родительских таблиц в результат включаются данные дочерней таблицы.

Когда в нескольких родительских таблицах оказываются столбцы с одним именем, происходит ошибка, за исключением случая, когда типы данных всех этих столбцов в таблицах совпадают. В этом случае одноимённые столбцы объединяются и формируют один столбец в новой таблице. Если имя столбца новой таблицы совпадает с именем одного из унаследованных столбцов, их типы так же должны совпадать, и в этом случае определения столбцов тоже сливаются в одну. Если в новой таблице явно указывается значение по умолчанию для нового столбца, это значение переопределяет любые значения по умолчанию, унаследованные этим столбцом. В противном случае, если значения по умолчанию определяются в разных родительских таблицах, эти определения должны совпадать, иначе произойдёт ошибка.

Ограничения `CHECK` объединяются вместе по сути так же, как и столбцы: если несколько родительских таблиц и/или определение новой таблицы содержат одноимённые ограничения `CHECK`, этим ограничениям должны соответствовать одинаковые выражения проверки, в

противном случае произойдёт ошибка. В случае совпадения выражения, эти ограничения с данным выражением объединяются в одно. При этом ограничения со свойством NO INHERIT в родительской таблице исключаются из рассмотрения. Заметьте, что безымянное ограничение CHECK в новой таблице никогда не сливается с другими, так как для него всегда выбирается уникальное имя.

Параметры STORAGE для столбца так же копируются из родительских таблиц.

Если столбец в родительской таблице является столбцом идентификации, это свойство не наследуется. Если требуется, в дочерней таблице этот столбец можно объявить столбцом идентификации.

PARTITION BY { RANGE | LIST } ({ *имя_столбца* | (*выражение*) } [*класс_операторов*] [, ...])

Необязательное предложение PARTITION BY задаёт стратегию секционирования таблицы. Таблица, созданная с этим указанием, называется *секционируемой* таблицей. Задаваемый в скобках список столбцов или выражений формирует *ключ разбиения* таблицы. Для разбиения по диапазонам ключ разбиения может включать несколько столбцов или выражений (до 32, но этот предел можно изменить при сборке Postgres Pro), но для разбиения по спискам ключ разбиения должен состоять из одного столбца или выражения. Если при создании секционируемой таблицы класс операторов B-дерева не указывается, будет использоваться класс операторов B-дерева по умолчанию для данного типа данных. Если такой класс операторов не находится, выдаётся ошибка.

Секционируемая таблица разделяется на подтаблицы (называемые секциями), которые создаются отдельными командами CREATE TABLE. Сама по себе секционируемая таблица не содержит данных. Строка данных, вставляемая в эту таблицу, перенаправляется в секцию в зависимости от значений столбцов или выражений в ключе разбиения. Если значениям в новой строке не соответствует ни одна из существующих секций, возникает ошибка.

Секционируемые таблицы не поддерживают ограничения UNIQUE, PRIMARY KEY, EXCLUDE и FOREIGN KEY; однако вы можете определить эти ограничения в отдельных секциях.

LIKE *исходная_таблица* [*вариант_копирования* ...]

Предложение LIKE определяет таблицу, из которой в новую таблицу будут автоматически скопированы все имена столбцов, их типы данных и их ограничения на NULL.

В отличие от INHERITS, новая и исходная таблица становятся полностью независимыми после завершения создания. Изменения в исходной таблице не отражаются в новой, а данные новой таблицы не включаются в результат чтения исходной.

Выражения значений по умолчанию в определениях копируемых столбцов переносятся в новую таблицу, только если дополнительно указано INCLUDING DEFAULTS. Без этого указания выражения по умолчанию не копируются, вследствие чего в новой таблице скопированные столбцы получают значения по умолчанию NULL. Заметьте, что при копировании выражений по умолчанию, которые вызывают функции, модифицирующие БД, как например nextval, может образовываться функциональная связь исходной таблицы с новой.

Характеристика идентификации в определении скопированного столбца будет копироваться, только если в команде есть указание INCLUDING IDENTITY. Для каждого столбца идентификации в новой таблице создаётся новая последовательность, не зависящая от последовательностей, связанных со старой таблицей.

Ограничения NULL копируются в новую таблицу всегда, а ограничения CHECK копируются только при явном указании INCLUDING CONSTRAINTS. Это в равной мере касается ограничений как таблиц, так и столбцов.

Расширенная статистика копируется в новую таблицу, если указано INCLUDING STATISTICS.

Индексы, свойство PRIMARY KEY, а также ограничения UNIQUE и EXCLUDE копируются из исходной таблицы в новую, только если явно указано INCLUDING INDEXES. Имена для новых индексов и ограничений выбираются согласно стандартным правилам, независимо от того, как назывались исходные. (Это позволяет избежать ошибок с дублированием имён с новыми индексами.)

Параметры STORAGE будут перенесены в определение копируемых столбцов, только если явно указывается INCLUDING STORAGE. По умолчанию параметры STORAGE исключаются, вследствие чего скопированные столбцы в новой таблице получают параметры по умолчанию, определённые соответствующим типом. Подробнее параметры STORAGE описаны в Разделе 63.2.

Комментарии для скопированных столбцов, ограничений и индексов будут копироваться, только если явно указано INCLUDING COMMENTS. По умолчанию комментарии исключаются, вследствие чего скопированные столбцы и ограничения в новой таблице оказываются без комментариев.

INCLUDING ALL представляет собой краткую форму предложения INCLUDING COMMENTS INCLUDING CONSTRAINTS INCLUDING DEFAULTS INCLUDING IDENTITY INCLUDING INDEXES INCLUDING STATISTICS INCLUDING STORAGE.

Заметьте, что в отличие от INHERITS, столбцы и ограничения, копируемые командой LIKE, не объединяются с одноимёнными столбцами и ограничениями. Если дублирующееся имя указывается явно или возникает в другом предложении LIKE, происходит ошибка.

Предложение LIKE может также применяться для копирования определений столбцов из представлений, сторонних таблиц и составных типов. Неприменимые параметры (например, INCLUDING INDEXES для представления) при этом игнорируются.

CONSTRAINT *имя_ограничения*

Необязательное имя столбца или ограничения таблицы. При нарушении ограничения его имя будет выводиться в сообщении об ошибках, так что имена ограничений вида столбец должен быть положительным могут сообщить полезную информацию об ограничении клиентскому приложению. (Имена ограничений, включающие пробелы, необходимо заключать в двойные кавычки.) Если имя ограничения не указано, система генерирует имя автоматически.

NOT NULL

Данный столбец не принимает значения NULL.

NULL

Данный столбец может содержать значения NULL (по умолчанию).

Это предложение предназначено только для совместимости с нестандартными базами данных SQL. Использовать его в новых приложениях не рекомендуется.

CHECK (*выражение*) [NO INHERIT]

В ограничении CHECK задаётся выражение, возвращающее логический результат, по которому определяется, будет ли успешна операция добавления или изменения для конкретных строк. Операция выполняется успешно, если результат выражения равен TRUE или UNKNOWN. Если же для какой-нибудь строки, задействованной в операции добавления или изменения, будет получен результат FALSE, возникает ошибка, и эта операция не меняет ничего в базе данных. Ограничение-проверка, заданное как ограничение столбца, должно ссылаться только на значение самого столбца, тогда как ограничение на уровне таблицы может ссылаться и на несколько столбцов.

В настоящее время выражения CHECK не могут содержать подзапросы или ссылаться на какие-либо переменные, кроме как на столбцы текущей строки (см. Подраздел 5.3.1). Также допустима ссылка на системный столбец tableoid, но не на другие системные столбцы.

Ограничение с пометкой NO INHERIT не будет наследоваться дочерними таблицами.

Когда для таблицы задано несколько ограничений CHECK, они будут проверяться для каждой строки в алфавитном порядке имён после проверки ограничений NOT NULL. (До версии 9.5 в PostgreSQL не было установлено никакого определённого порядка обработки ограничений CHECK.)

DEFAULT *выражение_по_умолчанию*

Предложение DEFAULT задаёт значение по умолчанию для столбца, в определении которого оно присутствует. Значение задаётся выражением без переменных (подзапросы и перекрёстные ссылки на другие столбцы текущей таблицы в нём не допускаются). Тип данных выражения, задающего значение по умолчанию, должен соответствовать типу данных столбца.

Это выражение будет использоваться во всех операциях добавления данных, в которых не задаётся значение данного столбца. Если значение по умолчанию не определено, таким значением будет NULL.

GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [(*параметры_последовательности*)]

С этим предложением столбец создаётся как *столбец идентификации*. С ним будет связана неявная последовательность, из которой этот столбец будет автоматически получать значения в новых строках. Такому столбцу неявно назначается свойство NOT NULL.

Предложения ALWAYS и BY DEFAULT определяют, какой приоритет имеет заданное пользователем значение этого столбца в команде INSERT. С предложением ALWAYS пользовательское значение используется, только если в операторе INSERT указано OVERRIDING SYSTEM VALUE. С предложением BY DEFAULT пользовательскому значению отдаётся предпочтение. За подробностями обратитесь к INSERT. (В команде COPY всегда используются пользовательские значения, вне зависимости от значения этого параметра.)

Используя необязательное предложение *параметры_последовательности*, можно переопределить свойства последовательности. За подробностями обратитесь к CREATE SEQUENCE.

UNIQUE *параметры_индекса* (ограничение столбца)
UNIQUE (*имя_столбца* [, ...]) *параметры_индекса* (ограничение таблицы)

Ограничение UNIQUE определяет, что группа из одного или нескольких столбцов таблицы может содержать только уникальные значения. Ограничение уникальности для таблицы ведёт себя точно так же, как ограничение для столбца, но может охватывать несколько столбцов. Таким образом, ограничение уникальности гарантирует, что любые две строки таблицы различаются как минимум в этих столбцах.

При проверке ограничения уникальности значения NULL не считаются равными.

В каждом ограничении уникальности должен задаваться набор столбцов, отличный от набора любого другого ограничения уникальности или первичного ключа в данной таблице. (Избыточные ограничения уникальности будут просто игнорироваться.)

При добавлении ограничения уникальности автоматически создаётся уникальный индекс-B-дерево по столбцу или группе столбцов, фигурирующих в ограничении. Дополнительные свойства индекса можно изменить, добавив предложения INCLUDE, WITH и USING INDEX TABLESPACES, как описано ниже.

PRIMARY KEY *параметры_индекса* (ограничение столбца)
PRIMARY KEY (*имя_столбца* [, ...]) *параметры_индекса* (ограничение таблицы)

Ограничение PRIMARY KEY определяет, что столбец или столбцы таблицы могут содержать только уникальные (без повторений) значения, отличные от NULL. Для таблицы может быть задан только один первичный ключ, будь то ограничение столбца или ограничение таблицы.

В определении первичного ключа должен задаваться набор столбцов, отличный от набора любого другого ограничения уникальности, установленного для данной таблицы. (В противном случае уникальное ограничение оказывается избыточным и будет отброшено.)

`PRIMARY KEY` устанавливает для данных те же ограничения, что и сочетание `UNIQUE` и `NOT NULL`, но созданный по набору столбцов первичный ключ также даёт метаинформацию о конструкции схемы, так как он подразумевает, что другие таблицы могут ссылаться на этот набор столбцов как на уникальный идентификатор строк.

При добавлении ограничения `PRIMARY KEY` автоматически создаётся уникальный индекс B-дерево по столбцу или группе столбцов, фигурирующих в ограничении. Дополнительные свойства индекса можно изменить, добавив предложения `INCLUDE`, `WITH` и `USING INDEX TABLESPACES`, как описано ниже.

`EXCLUDE [ USING` *`индексный_метод`* `] ( `*`элемент_исключения`* `WITH` *`оператор`* `[, ... ] )` *`параметры_индекса`* `[ WHERE (` *`предикат`* `) ]`

Предложение `EXCLUDE` определяет ограничение-исключение, которое гарантирует, что для любых двух строк, сравниваемых по указанным столбцам или выражениям с указанными операторами, результат не будет равен `TRUE` для всех сравнений. Если все указанные операторы проверяют равенство, это ограничение равносильно ограничению `UNIQUE`, хотя обычное ограничение уникальности будет работать быстрее. С другой стороны, в ограничениях-исключениях можно задавать более общие условия, чем простое условие на равенство. Например, можно задать ограничение, требующее, чтобы никакие две строки в таблице не содержали пересекающихся кругов (см. Раздел 8.8), применив оператор `&&`.

Ограничения-исключения реализуются с помощью индексов, так что каждый указанный в них оператор должен быть связан с соответствующим классом операторов (см. Раздел 11.9) для *`метода_индекса`*. Кроме того, операторы должны быть коммутативными. В каждом *`элементе_исключения`* можно дополнительно указать класс оператора и/или параметры сортировки, подробно описанные в CREATE INDEX. Вы также можете изменить свойства индекса, добавив предложения `INCLUDE`, `WITH` и `USING INDEX TABLESPACES`, как описано ниже.

Индексный метод доступа должен поддерживать `amgettuple` (см. Главу 56); в настоящее время это означает, что индексы GIN для этого не подходят. Хотя в ограничении-исключении можно использовать B-деревья и хеш-индексы, в этом мало смысла, так как такой подход ничем не лучше обычного ограничения уникальности. Так что на практике методом доступа всегда будет GiST или SP-GiST.

Параметр *`предикат`* позволяет указать ограничение-исключение для подмножества таблицы; внутри при этом создаётся частичный индекс. Заметьте, что предикат необходимо заключить в скобки.

`REFERENCES` *`внешняя_таблица`* `[ (` *`внешний_столбец`* `) ] [ MATCH` *`тип_совпадения`* `] [ ON DELETE` *`действие`* `] [ ON UPDATE` *`действие`* `]` (ограничение столбца)
`FOREIGN KEY (` *`имя_столбца`* `[, ... ] ) REFERENCES` *`внешняя_таблица`* `[ (` *`внешний_столбец`* `[, ... ] ) ] [ MATCH` *`тип_совпадения`* `] [ ON DELETE` *`действие`* `] [ ON UPDATE` *`действие`* `]` (ограничение таблицы)

Эти предложения определяют ограничение внешнего ключа, требующее, чтобы группа из одного или нескольких столбцов новой таблицы содержала только такие значения, которым соответствуют значения в заданных столбцах некоторой строки во внешней таблице. Если список *`целевых_столбцов`* опущен, в качестве него используется первичный ключ *`целевой_таблицы`*. В качестве целевых столбцов должны указываться столбцы неоткладываемого уникального ограничения или первичного ключа во внешней таблице. При этом пользователь должен иметь право `REFERENCES` во внешней таблице (либо для всей таблицы, либо только для целевых столбцов). Заметьте, что ограничения внешнего ключа, связывающие временные и постоянные таблицы, определить нельзя.

Значения, вставляемые в ссылающиеся столбцы, сверяются со значениями во внешних столбцах внешней таблицы с учётом заданного типа совпадения. Возможны три типа

совпадения: MATCH FULL (полное совпадение), MATCH PARTIAL (частичное совпадение) и тип по умолчанию, MATCH SIMPLE (простое совпадение). С MATCH FULL ни один из столбцов составного внешнего ключа не может содержать NULL, кроме случая, когда все внешние столбцы NULL; в этом случае строка может не иметь соответствия во внешней таблице. С MATCH SIMPLE любой из столбцов внешнего ключа может содержать NULL; при этом строка с NULL в одном из таких столбцов может не иметь соответствия во внешней таблице. Тип MATCH PARTIAL ещё не реализован. (Разумеется, чтобы вопросы со сравнением NULL не возникали, к столбцам, ссылающимся на внешние, можно применить ограничения NOT NULL.)

Кроме того, при изменении значений во внешних столбцах с данными в столбцах этой таблицы могут производиться определённые действия. Предложение ON DELETE задаёт действие, производимое при удалении некоторой строки во внешней таблице. Предложение ON UPDATE подобным образом задаёт действие, производимое при изменении значения в целевых столбцах внешней таблицы. Если строка изменена, но это изменение не затронуло целевые столбцы, никакое действие не производится. Ссылочные действия, кроме NO ACTION, нельзя сделать откладываемыми, даже если ограничение объявлено как откладываемое. Для каждого предложения возможные следующие варианты действий:

NO ACTION

Выдать ошибку, показывающую, что при удалении или изменении записи произойдёт нарушение ограничения внешнего ключа. Для отложенных ограничений ошибка произойдёт в момент проверки ограничения, если строки, ссылающиеся на эту запись, по-прежнему будут существовать. Этот вариант действия подразумевается по умолчанию.

RESTRICT

Выдать ошибку, показывающую, что при удалении или изменении записи произойдёт нарушение ограничения внешнего ключа. Этот вариант подобен NO ACTION, но эта проверка будет неоткладываемой.

CASCADE

Удалить все строки, ссылающиеся на удаляемую запись, либо поменять значения в ссылающихся столбцах на новые значения во внешних столбцах, в соответствии с операцией.

SET NULL

Установить ссылающиеся столбцы равными NULL.

SET DEFAULT

Установить в ссылающихся столбцах значения по умолчанию. (Если эти значения не равны NULL, во внешней таблице должна быть строка, соответствующая набору значений по умолчанию; в противном случае операция завершится ошибкой.)

Если внешние столбцы меняются часто, будет разумным добавить индекс для ссылающихся столбцов, чтобы действия по обеспечению ссылочной целостности, связанные с ограничением внешнего ключа, выполнялись более эффективно.

DEFERRABLE
NOT DEFERRABLE

Это предложение определяет, может ли ограничение быть отложенным. Неоткладываемое ограничение будет проверяться немедленно после каждой команды. Проверка откладываемых ограничений может быть отложена до завершения транзакции (обычно с помощью команды SET CONSTRAINTS). По умолчанию подразумевается вариант NOT DEFERRABLE. В настоящее время это предложение принимают только ограничения UNIQUE, PRIMARY KEY, EXCLUDE и REFERENCES (внешний ключ). Ограничения NOT NULL и CHECK не могут быть отложенными. Заметьте, что откладываемые ограничения не могут применяться в качестве решающих при конфликте в операторе INSERT с предложением ON CONFLICT DO UPDATE.

INITIALLY IMMEDIATE
INITIALLY DEFERRED

Для откладываемых ограничений это предложение определяет, когда ограничение должно проверяться по умолчанию. Ограничение с характеристикой INITIALLY IMMEDIATE (подразумеваемой по умолчанию) проверяется после каждого оператора. Ограничение INITIALLY DEFERRED, напротив, проверяется только в конце транзакции. Время проверки ограничения можно изменить явно с помощью команды SET CONSTRAINTS.

INCLUDE (*имя_столбца* [, ...])

Необязательное предложение INCLUDE добавляет неключевые столбцы в индекс, создаваемый для ограничений UNIQUE, PRIMARY KEY или EXCLUDE; данные ограничения при этом не будут распространятся на эти столбцы. Неключевые столбцы полезны тем, что их содержимое можно получить при сканировании только индекса. Используя данное предложение, в связанный с ограничением индекс можно включить столбцы, часто фигурирующие в запросах. Заметьте, что хотя ограничение по неключевым столбцам не контролируется, оно всё же зависит от них. Как следствие, некоторые операции с этими столбцами (например, DROP COLUMN) могут повлечь каскадное удаление ограничения и индекса. Более подробно о неключевых столбцах рассказывается в описании предложения INCLUDE команды CREATE INDEX.

WITH (*параметр_хранения* [= *значение*] [, ...])

Это предложение определяет необязательные параметры хранения для таблицы или индекса (за подробными сведениями о них обратитесь к Подразделу «Параметры хранения»). Предложение WITH для таблицы может также включать указание OIDS=TRUE (или просто OIDS), устанавливающее, что каждая строка таблицы должна иметь собственный OID (Object IDentifier, идентификатор объекта), или указание OIDS=FALSE, устанавливающее, что строки не содержат OID. Если указание OIDS отсутствует, значение этого свойства по умолчанию зависит от конфигурационного параметра default_with_oids. (Если новая таблица унаследована от каких-либо таблиц, имеющих OID, свойство OIDS=TRUE задаётся принудительно, даже если в команде явно написано OIDS=FALSE.)

Если явно указано OIDS=FALSE или это подразумевается, в новой таблице не будут храниться значения OID и новый OID не будет генерироваться для каждой добавляемой в неё строки. Для обычных таблиц такое поведение предпочтительнее, так как оно сокращает потребление OID и тем самым откладывает зацикливание 32-битного счётчика OID. Как только происходит зацикливание, значения OID больше нельзя считать уникальными, что делает их значительно менее полезными. К тому же, исключение столбца OID из таблицы сокращает объём, необходимый для хранения таблицы на диске, на 4 байта для каждой строки (на большинстве платформ), что несколько улучшает производительность.

Для удаления данных OID из таблицы после её создания воспользуйтесь командой ALTER TABLE.

WITH OIDS
WITHOUT OIDS

Это устаревшее написание указаний WITH (OIDS) и WITH (OIDS=FALSE), соответственно. Если требуется определить одновременно свойство OIDS и параметры хранения, необходимо использовать синтаксис WITH (...); см. ниже.

ON COMMIT

Поведением временных таблиц в конце блока транзакции позволяет управлять предложение ON COMMIT, которое принимает три параметра:

PRESERVE ROWS

Никакое специальное действие в конце транзакции не выполняется. Это поведение по умолчанию.

DELETE ROWS

Все строки в этой временной таблице будут удаляться в конце каждого блока транзакции. По сути, при каждой фиксации транзакции будет автоматически выполняться TRUNCATE. В случае секционированной таблицы это действие не распространяется на её секции.

DROP

Временная таблица будет удалена в конце текущего блока транзакции. Если это секционированная таблица, будут удалены и все её секции. Если у таблицы есть потомки в иерархии наследования, они также будут удалены.

TABLESPACE *табл_пространство*

Здесь *табл_пространство* — имя табличного пространства, в котором будет создаваться новая таблица. Если оно не указано, выбирается default_tablespace или temp_tablespaces, если таблица временная.

USING INDEX TABLESPACE *табл_пространство*

Это предложение позволяет выбрать табличное пространство, в котором будут создаваться индексы, связанные с ограничениями UNIQUE, PRIMARY KEY или EXCLUDE. Если оно не указано, выбирается default_tablespace или temp_tablespaces, если таблица временная.

Параметры хранения

Предложение WITH позволяет установить *параметры хранения* для таблиц и индексов, связанных с ограничениями UNIQUE, PRIMARY KEY и EXCLUDE. Параметры хранения для индексов документированы в CREATE INDEX. Поддерживаемые в настоящее время параметры хранения для таблиц перечислены ниже. Как показано, для многих параметров существует дополнительный параметр с тем же именем и префиксом toast., который управляет поведением вторичной таблицы TOAST, если она есть (за дополнительными сведениями о TOAST обратитесь к Разделу 63.2). Если значение некоторого параметра задано для таблицы, а значение равнозначного параметра toast. не определено, для таблицы TOAST будет применяться значение параметра основной таблицы. Возможность задания этих параметров для секционированных таблиц не поддерживается, но вы можете задать их для отдельных конечных секций.

fillfactor (integer)

Фактор заполнения для таблицы, задаваемый в процентах, от 10 до 100. Значение по умолчанию — 100 (плотное заполнение). При меньшем факторе заполнения операции INSERT упаковывают данные в страницы только до заданного процента; оставшееся место резервируется для изменения строк на этой странице. В результате UPDATE получает шанс поместить изменённую копию строки в ту же страницу, что и исходную, что гораздо эффективнее, чем размещать её на другой странице. Для таблиц, записи в которых никогда не меняются, лучшим выбором будет плотное заполнение, но для активно изменяемых таблиц лучше выбрать меньший фактор заполнения. Этот параметр нельзя задать для таблиц TOAST.

parallel_workers (integer)

Данный параметр задаёт число рабочих процессов, которые должны задействоваться при параллельном сканировании таблицы. Если это значение не задано, система будет определять его, исходя из размера отношения. Фактическое число рабочих процессов может быть меньше, например, вследствие ограничения max_worker_processes.

autovacuum_enabled, toast.autovacuum_enabled (boolean)

Включает или отключает демон автоочистки для определённой таблицы. Со значением true демон автоочистки будет автоматически выполнять операции VACUUM и/или ANALYZE в этой таблице, согласно правилам, описанным в Подразделе 23.1.6. Со значением false эта таблица не будет подвергаться автоочистке, если только это не потребуется для предотвращения зацикливания идентификаторов транзакций. Более подробно предотвращение зацикливания описывается в Подразделе 23.1.5. Заметьте, что демон автоочистки не будет запускаться

вовсе (если только это не потребуется для предотвращения зацикливания), если параметр autovacuum имеет значение false; это нельзя переопределить, установив параметры хранения для отдельных таблиц. Таким образом, явно устанавливать для этого параметра значение `true` практически не имеет смысла — полезно только значение `false`.

`autovacuum_vacuum_threshold`, `toast.autovacuum_vacuum_threshold` (`integer`)

Значение параметра autovacuum_vacuum_threshold для таблицы.

`autovacuum_vacuum_scale_factor`, `toast.autovacuum_vacuum_scale_factor` (`floating point`)

Значение параметра autovacuum_vacuum_scale_factor для таблицы.

`autovacuum_analyze_threshold` (`integer`)

Значение параметра autovacuum_analyze_threshold для таблицы.

`autovacuum_analyze_scale_factor` (`floating point`)

Значение параметра autovacuum_analyze_scale_factor для таблицы.

`autovacuum_vacuum_cost_delay`, `toast.autovacuum_vacuum_cost_delay` (`integer`)

Значение параметра autovacuum_vacuum_cost_delay для таблицы.

`autovacuum_vacuum_cost_limit`, `toast.autovacuum_vacuum_cost_limit` (`integer`)

Значение параметра autovacuum_vacuum_cost_limit для таблицы.

`autovacuum_freeze_min_age`, `toast.autovacuum_freeze_min_age` (`integer`)

Значение параметра vacuum_freeze_min_age для таблицы. Учтите, что система будет игнорировать установленные для таблиц значения `autovacuum_freeze_min_age`, превышающие половину системного autovacuum_freeze_max_age.

`autovacuum_freeze_max_age`, `toast.autovacuum_freeze_max_age` (`integer`)

Значение параметра autovacuum_freeze_max_age для таблицы. Учтите, что система будет игнорировать установленные для таблиц значения `autovacuum_freeze_max_age`, превышающие значение системного параметра (они могут быть только меньше).

`autovacuum_freeze_table_age`, `toast.autovacuum_freeze_table_age` (`integer`)

Значение параметра vacuum_freeze_table_age для таблицы.

`autovacuum_multixact_freeze_min_age`, `toast.autovacuum_multixact_freeze_min_age` (`integer`)

Значение параметра vacuum_multixact_freeze_min_age для таблицы. Учтите, что демон автоочистки будет игнорировать установленные для таблиц значения `autovacuum_multixact_freeze_min_age`, превышающие половину значения системного параметра autovacuum_multixact_freeze_max_age.

`autovacuum_multixact_freeze_max_age`, `toast.autovacuum_multixact_freeze_max_age` (`integer`)

Значение параметра autovacuum_multixact_freeze_max_age для таблицы. Учтите, что система автоочистки будет игнорировать установленные для таблиц параметры `autovacuum_multixact_freeze_max_age`, превышающие системный параметр (они могут быть только меньше).

`autovacuum_multixact_freeze_table_age`, `toast.autovacuum_multixact_freeze_table_age` (`integer`)

Значения параметра vacuum_multixact_freeze_table_age для таблицы.

`log_autovacuum_min_duration`, `toast.log_autovacuum_min_duration` (`integer`)

Значения параметра log_autovacuum_min_duration для таблицы.

`user_catalog_table` (`boolean`)

Объявляет таблицу как дополнительную таблицу каталога, например для целей логической репликации. За подробностями обратитесь к Подразделу 46.6.2. Для таблиц TOAST этот параметр задать нельзя.

Замечания

Применять OID в новых приложениях не рекомендуется: по возможности лучше использовать в качестве первичного ключа таблицы столбец идентификации или другой генератор последовательности. Однако если в вашем приложении для идентификации строк применяется OID, рекомендуется создать уникальное ограничение по столбцу `oid` в этой таблице, чтобы значения OID в этой таблице на самом деле однозначно идентифицировали строки даже после зацикливания счётчика. Также не стоит полагать, что значения OID уникальны в разных таблицах; если вам требуется идентификатор, уникальный в базе данных, воспользуйтесь для этого комбинацией `tableoid` и OID строки.

Подсказка

Применять `OIDS=FALSE` не рекомендуется для таблиц без первичного ключа, так как без OID или уникального ключа данных сложно идентифицировать определённые строки.

Postgres Pro автоматически создаёт индекс, гарантирующий уникальность, для каждого ограничения уникальности и ограничения первичного ключа. Поэтому явно создавать индекс для столбцов первичного ключа не требуется. (За дополнительными сведениями обратитесь к CREATE INDEX.)

Ограничения уникальности и первичные ключи в текущей реализации не наследуются. Вследствие этого ограничения уникальности довольно плохо сочетаются с наследованием.

В таблице не может быть больше 1600 столбцов. (На практике фактический предел обычно ниже из-за ограничения на длину записи.)

Примеры

Создание таблицы `films` и таблицы `distributors`:

```
CREATE TABLE films (
    code        char(5) CONSTRAINT firstkey PRIMARY KEY,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
);

CREATE TABLE distributors (
     did    integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
     name   varchar(40) NOT NULL CHECK (name <> '')
);
```

Создание таблицы с двумерным массивом:

```
CREATE TABLE array_int (
    vector  int[][]
);
```

Определение ограничения уникальности для таблицы `films`. Ограничения уникальности могут быть определены для одного или нескольких столбцов таблицы:

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT production UNIQUE(date_prod)
);
```

Определение ограничения-проверки для столбца:

```
CREATE TABLE distributors (
    did     integer CHECK (did > 100),
    name    varchar(40)
);
```

Определение ограничения-проверки для таблицы:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    CONSTRAINT con1 CHECK (did > 100 AND name <> '')
);
```

Определение ограничения первичного ключа для таблицы `films`:

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT code_title PRIMARY KEY(code,title)
);
```

Определение ограничения первичного ключа для таблицы `distributors`. Следующие два примера равнозначны, но в первом используется синтаксис ограничений для таблицы, а во втором — для столбца:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    PRIMARY KEY(did)
);

CREATE TABLE distributors (
    did     integer PRIMARY KEY,
    name    varchar(40)
);
```

Определение значений по умолчанию: для столбца `name` значением по умолчанию будет строка, для столбца `did` — следующее значение объекта последовательности, а для `modtime` — время, когда была вставлена запись:

```
CREATE TABLE distributors (
    name      varchar(40) DEFAULT 'Luso Films',
    did       integer DEFAULT nextval('distributors_serial'),
    modtime   timestamp DEFAULT current_timestamp
);
```

Определение двух ограничений `NOT NULL` для столбцов таблицы `distributors`, при этом одному ограничению даётся явное имя:

```
CREATE TABLE distributors (
    did     integer CONSTRAINT no_null NOT NULL,
    name    varchar(40) NOT NULL
);
```

Определение ограничения уникальности для столбца `name`:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40) UNIQUE
);
```

То же самое условие, но в виде ограничения таблицы:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name)
);
```

Создание такой же таблицы с фактором заполнения 70% для таблицы и её уникального индекса:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name) WITH (fillfactor=70)
)
WITH (fillfactor=70);
```

Создание таблицы `circles` с ограничением-исключением, не допускающим пересечения двух кругов:

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

Создание таблицы `cinemas` в табличном пространстве `diskvol1`:

```
CREATE TABLE cinemas (
        id serial,
        name text,
        location text
) TABLESPACE diskvol1;
```

Создание составного типа и типизированной таблицы:

```
CREATE TYPE employee_type AS (name text, salary numeric);

CREATE TABLE employees OF employee_type (
    PRIMARY KEY (name),
    salary WITH OPTIONS DEFAULT 1000
);
```

Создание таблицы, секционируемой по диапазонам:

```
CREATE TABLE measurement (
    logdate         date not null,
    peaktemp        int,
    unitsales       int
```

```
) PARTITION BY RANGE (logdate);
```

Создание таблицы, секционируемой по диапазонам, с ключом разбиения, включающим несколько столбцов:

```
CREATE TABLE measurement_year_month (
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (EXTRACT(YEAR FROM logdate), EXTRACT(MONTH FROM logdate));
```

Создание таблицы, секционируемой по спискам:

```
CREATE TABLE cities (
    city_id      bigserial not null,
    name         text not null,
    population   bigint
) PARTITION BY LIST (left(lower(name), 1));
```

Создание секции таблицы, секционируемой по диапазонам:

```
CREATE TABLE measurement_y2016m07
    PARTITION OF measurement (
    unitsales DEFAULT 0
) FOR VALUES FROM ('2016-07-01') TO ('2016-08-01');
```

Создание нескольких секций для таблицы, секционируемой по диапазонам, с ключом разбиения, включающим несколько столбцов:

```
CREATE TABLE measurement_ym_older
    PARTITION OF measurement_year_month
    FOR VALUES FROM (MINVALUE, MINVALUE) TO (2016, 11);

CREATE TABLE measurement_ym_y2016m11
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2016, 11) TO (2016, 12);

CREATE TABLE measurement_ym_y2016m12
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2016, 12) TO (2017, 01);

CREATE TABLE measurement_ym_y2017m01
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2017, 01) TO (2017, 02);
```

Создание секции таблицы, секционируемой по спискам:

```
CREATE TABLE cities_ab
    PARTITION OF cities (
    CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('a', 'b');
```

Создание секции таблицы, секционируемой по спискам (при этом сама секция также создаётся секционируемой), и добавление секции в неё:

```
CREATE TABLE cities_ab
    PARTITION OF cities (
    CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('a', 'b') PARTITION BY RANGE (population);

CREATE TABLE cities_ab_10000_to_100000
    PARTITION OF cities_ab FOR VALUES FROM (10000) TO (100000);
```

Совместимость

Команда CREATE TABLE соответствует стандарту SQL, с описанными ниже исключениями.

Временные таблицы

Хотя синтаксис CREATE TEMPORARY TABLE подобен аналогичному в стандарте SQL, результат получается другим. В стандарте временные таблицы определяются только один раз и существуют (изначально пустые) в каждом сеансе, в котором они используются. Postgres Pro вместо этого требует, чтобы каждый сеанс выполнял собственную команду CREATE TEMPORARY TABLE для каждой временной таблицы, которая будет использоваться. Это позволяет использовать в разных сеансах таблицы с одинаковыми именами для разных целей, тогда как при подходе, регламентированном стандартом, все экземпляры временной таблицы с одним именем должны иметь одинаковую табличную структуру.

Поведение временных таблиц, описанное в стандарте, в большинстве своём игнорируют и другие СУБД, так что в этом отношении Postgres Pro ведёт себя так же, как и ряд других СУБД.

В стандарте SQL также разделяются глобальные и локальные временные таблицы — в локальной временной таблице содержится отдельный набор данных для каждого модуля SQL в отдельном сеансе, хотя её определение так же разделяется между ними. Так как в Postgres Pro модули SQL не поддерживаются, это различие в Postgres Pro не существует.

Совместимости ради, Postgres Pro принимает ключевые слова GLOBAL и LOCAL в объявлении временной таблицы, но в настоящее время они никак не действуют. Использовать их не рекомендуется, так как в будущих версиях Postgres Pro может быть принята их интерпретация, более близкая к стандарту.

Предложение ON COMMIT для временных таблиц тоже подобно описанному в стандарте SQL, но есть некоторые отличия. Если предложение ON COMMIT опущено, в SQL подразумевается поведение ON COMMIT DELETE ROWS. Однако в Postgres Pro по умолчанию действует ON COMMIT PRESERVE ROWS. Параметр ON COMMIT DROP в стандарте SQL отсутствует.

Неотложенные ограничения уникальности

Когда ограничение UNIQUE или PRIMARY KEY не является отложенным, Postgres Pro проверяет уникальность непосредственно в момент добавления или изменения строки. Стандарт SQL говорит, что уникальность должна обеспечиваться только в конце оператора; это различие проявляется, например когда одна команда изменяет множество ключевых значений. Чтобы получить поведение, оговоренное стандартом, объявите ограничение как откладываемое (DEFERRABLE), но не отложенное (т. е., INITIALLY IMMEDIATE). Учтите, что этот вариант может быть значительно медленнее, чем немедленная проверка ограничений.

Ограничения-проверки для столбцов

Стандарт SQL говорит, что ограничение CHECK, определяемое для столбца, может ссылаться только на столбец, с которым оно связано; только ограничения CHECK для таблиц могут ссылаться на несколько столбцов. В Postgres Pro этого ограничения нет; он воспринимает ограничения-проверки для столбцов и таблиц одинаково.

Ограничение EXCLUDE

Ограничения EXCLUDE являются расширением Postgres Pro.

NULL «Ограничение»

«Ограничение» NULL (на самом деле это не ограничение) является расширением Postgres Pro стандарта SQL, которое реализовано для совместимости с некоторыми другими СУБД (и для симметрии с ограничением NOT NULL). Так как это поведение по умолчанию для любого столбца, его присутствие не несёт смысловой нагрузки.

Наследование

Множественное наследование посредством `INHERITS` является языковым расширением Postgres Pro. SQL:1999 и более поздние стандарты определяют единичное наследование с другим синтаксисом и смыслом. Наследование в стиле SQL:1999 пока ещё не поддерживается в Postgres Pro.

Таблицы с нулём столбцов

Postgres Pro позволяет создать таблицу без столбцов (например, `CREATE TABLE foo();`). Это расширение стандарта SQL, который не допускает таблицы с нулём столбцов. Таблицы с нулём столбцов сами по себе не очень полезны, но если их запретить, возникают странные особые ситуации с командой `ALTER TABLE DROP COLUMN`, так что лучшим вариантом кажется игнорировать это требование стандарта.

Множество столбцов идентификации

Postgres Pro позволяет иметь в таблице более одного столбца идентификации. В стандарте же говорится, что в таблице может быть максимум один столбец идентификации. Это ограничение ослаблено в основном для большей гибкости при выполнении изменений в схеме или миграции. Заметьте, что команда `INSERT` поддерживает только одно предложение переопределения значения, применяемое ко всему оператору, так что с несколькими столбцами идентификации различное поведение не поддерживается должным образом.

Предложение `LIKE`

Хотя предложение `LIKE` описано в стандарте SQL, многие варианты его использования, допустимые в Postgres Pro, в стандарте не описаны, а некоторые предусмотренные в стандарте возможности не реализованы в Postgres Pro.

Предложение `WITH`

Предложение `WITH` является расширением Postgres Pro; в стандарте ни параметры хранения, ни OID не оговариваются.

Табличные пространства

Концепция табличных пространств в Postgres Pro отсутствует в стандарте. Как следствие, предложения `TABLESPACE` и `USING INDEX TABLESPACE` являются расширениями.

Типизированные таблицы

Типизированные таблицы реализуют подмножество стандарта SQL. Согласно стандарту, типизированная таблица содержит столбцы, соответствующие нижележащему составному типу, и ещё один столбец, ссылающийся на себя. Postgres Pro не поддерживает ссылающиеся на себя столбцы явно, но тот же эффект можно получить, воспользовавшись OID.

Предложение `PARTITION BY`

Предложение `PARTITION BY` является расширением Postgres Pro.

Предложение `PARTITION OF`

Предложение `PARTITION OF` является расширением Postgres Pro.

См. также

ALTER TABLE, DROP TABLE, CREATE TABLE AS, CREATE TABLESPACE, CREATE TYPE

CREATE TABLE AS

CREATE TABLE AS — создать таблицу из результатов запроса

Синтаксис

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
 EXISTS ] имя_таблицы
    [ (имя_столбца [, ...] ) ]
    [ WITH ( параметр_хранения [= значение] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
    [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
    [ TABLESPACE табл_пространство ]
    AS запрос
    [ WITH [ NO ] DATA ]
```

Описание

`CREATE TABLE AS` создаёт таблицу и наполняет её данными, полученными в результате выполнения `SELECT`. Столбцы этой таблицы получают имена и типы данных в соответствии со столбцами результата `SELECT` (хотя имена столбцов можно переопределить, добавив явно список новых имён столбцов).

`CREATE TABLE AS` напоминает создание представления, но на самом деле есть значительная разница: эта команда создаёт новую таблицу и выполняет запрос только раз, чтобы наполнить таблицу начальными данными. Последующие изменения в исходных таблицах запроса в новой таблице отражаться не будут. С представлением, напротив, определяющая его команда `SELECT` выполняется при каждой выборке из него.

Параметры

`GLOBAL` или `LOCAL`

Для совместимости игнорируются. Использование этих ключевых слов считается устаревшим; за подробностями обратитесь к CREATE TABLE.

`TEMPORARY` или `TEMP`

Если указано, создаваемая таблица будет временной. За подробностями обратитесь к CREATE TABLE.

`UNLOGGED`

Если указано, создаваемая таблица будет нежурналируемой. За подробностями обратитесь к CREATE TABLE.

`IF NOT EXISTS`

Не считать ошибкой, если отношение с таким именем уже существует; просто выдать замечание и оставить таблицу без изменений.

`имя_таблицы`

Имя создаваемой таблицы (возможно, дополненное схемой).

`имя_столбца`

Имя столбца в создаваемой таблице. Если имена столбцов не заданы явно, они определяются по именам столбцов результата запроса.

`WITH ( параметр_хранения [= значение] [, ... ] )`

Это предложение определяет дополнительные параметры хранения для новой таблицы: за подробностями обратитесь к Подразделу «Параметры хранения». Предложение `WITH` может

также включать указание OIDS=TRUE (или просто OIDS), с которым строкам в новой таблице будут назначаться идентификаторы объектов (OID), либо указание OIDS=FALSE, с которым строки не будут содержать OID. За дополнительными сведениями обратитесь к CREATE TABLE.

WITH OIDS
WITHOUT OIDS

Это устаревшее написание указаний WITH (OIDS) и WITH (OIDS=FALSE), соответственно. Если требуется определить одновременно свойство OIDS и параметры хранения, необходимо использовать синтаксис WITH (...); см. ниже.

ON COMMIT

Поведением временных таблиц в конце блока транзакции позволяет управлять предложение ON COMMIT, которое принимает три параметра:

PRESERVE ROWS

Никакое специальное действие в конце транзакции не выполняется. Это поведение по умолчанию.

DELETE ROWS

Все строки в этой временной таблице будут удаляться в конце каждого блока транзакции. По сути, при каждой фиксации транзакции будет автоматически выполняться TRUNCATE.

DROP

Эта временная таблица будет удаляться в конце текущего блока транзакции.

TABLESPACE *табл_пространство*

Здесь *табл_пространство* — имя табличного пространства, в котором будет создаваться новая таблица. Если оно не указано, выбирается default_tablespace или temp_tablespaces, если таблица временная.

запрос

Команда SELECT, TABLE или VALUES, либо команда EXECUTE, выполняющая подготовленный запрос SELECT, TABLE или VALUES.

WITH [NO] DATA

Это предложение определяет, будут ли данные, выданные запросом, копироваться в новую таблицу. Если нет, то копируется только структура. По умолчанию данные копируются.

Замечания

Функциональность этой команды подобна SELECT INTO, но предпочтительнее использовать её, во избежание путаницы с другими применениями синтаксиса SELECT INTO. Кроме того, набор возможностей CREATE TABLE AS шире, чем у SELECT INTO.

Команда CREATE TABLE AS позволяет пользователю явно определить, добавлять ли OID в таблицу. Если присутствие OID не определено явно, оно определяется конфигурационной переменной default_with_oids.

Примеры

Создание таблицы films_recent, содержащей только последние записи из таблицы films:

```
CREATE TABLE films_recent AS
  SELECT * FROM films WHERE date_prod >= '2002-01-01';
```

Чтобы скопировать таблицу полностью, можно также использовать короткую форму команды TABLE:

```
CREATE TABLE films2 AS
  TABLE films;
```

Создание временной таблицы `films_recent`, содержащей только последние записи таблицы `films`, с применением подготовленного оператора. Новая таблица будет содержать OID и прекратит существование при фиксации транзакции:

```
PREPARE recentfilms(date) AS
  SELECT * FROM films WHERE date_prod > $1;
CREATE TEMP TABLE films_recent WITH (OIDS) ON COMMIT DROP AS
  EXECUTE recentfilms('2002-01-01');
```

Совместимость

`CREATE TABLE AS` соответствует стандарту SQL. Нестандартные расширения перечислены ниже:

- Стандарт требует заключать предложение подзапроса в скобки, но в Postgres Pro эти скобки необязательны.
- Стандарт требует наличия указания `WITH [ NO ] DATA`, в Postgres Pro оно необязательно.
- Postgres Pro работает с временными таблицами не так, как описано в стандарте; за подробностями обратитесь к CREATE TABLE.
- Предложение `WITH` является расширением Postgres Pro; в стандарте ни параметры хранения, ни OID не оговариваются.
- Концепция табличных пространств в Postgres Pro отсутствует в стандарте. Как следствие, предложение `TABLESPACE` является расширением.

См. также

CREATE MATERIALIZED VIEW, CREATE TABLE, EXECUTE, SELECT, SELECT INTO, VALUES

CREATE TABLESPACE

CREATE TABLESPACE — создать табличное пространство

Синтаксис

```
CREATE TABLESPACE табл_пространство
    [ OWNER { новый_владелец | CURRENT_USER | SESSION_USER } ]
    LOCATION 'каталог'
    [ WITH ( параметр_табличного_пространства = значение [, ... ] ) ]
```

Описание

`CREATE TABLESPACE` регистрирует новое табличное пространство на уровне кластера баз данных. Имя табличного пространства должно отличаться от имён уже существующих табличных пространств в кластере.

Табличные пространства позволяют суперпользователям определять альтернативные расположения в файловой системе, где могут находиться файлы, содержащие объекты базы данных (например, таблицы или индексы).

Пользователь, имеющий соответствующие права, может передать параметр *табл_пространство* команде `CREATE DATABASE`, `CREATE TABLE`, `CREATE INDEX` или `ADD CONSTRAINT`, чтобы файлы данных для этих объектов хранились в указанном табличном пространстве.

> **Предупреждение**
>
> Табличное пространство нельзя использовать отдельно от кластера, в котором оно было определено; см. Раздел 21.6.

Параметры

табл_пространство

Имя создаваемого табличного пространства. Это имя не может начинаться с `pg_`, так как такие имена зарезервированы для системных табличных пространств.

имя_пользователя

Имя пользователя, который будет владельцем табличного пространства. Если опущено, владельцем по умолчанию станет пользователь, выполняющий команду. Создавать табличные пространства могут только суперпользователи, но их владельцами могут быть назначены и обычные пользователи.

каталог

Каталог, который будет использован для этого табличного пространства. Этот каталог должен быть пустым и должен принадлежать системному пользователю Postgres Pro. Задаваться его расположение должно абсолютным путём.

параметр_табличного_пространства

Устанавливаемый или сбрасываемый параметр табличного пространства. В настоящее время поддерживаются только параметры `seq_page_cost`, `random_page_cost` и `effective_io_concurrency`. При установке этих значений для заданного табличного пространства будет переопределена обычная оценка стоимости чтения страниц из таблиц в этом пространстве, настраиваемая одноимённым параметром конфигурации (см. seq_page_cost, random_page_cost, effective_io_concurrency). Это может быть полезно, если одно

из табличных пространств размещено на диске, который быстрее или медленнее остальной дисковой подсистемы.

Замечания

Табличные пространства поддерживаются только на платформах, поддерживающих символические ссылки.

`CREATE TABLESPACE` не может быть выполнена внутри блока транзакции.

Примеры

Создание табличного пространства `dbspace` в `/data/dbs`:

```
CREATE TABLESPACE dbspace LOCATION '/data/dbs';
```

Создание табличного пространства `indexspace` в `/data/indexes` и назначение его владельцем `genevieve`:

```
CREATE TABLESPACE indexspace OWNER genevieve LOCATION '/data/indexes';
```

Совместимость

`CREATE TABLESPACE` является расширением Postgres Pro.

См. также

CREATE DATABASE, CREATE TABLE, CREATE INDEX, DROP TABLESPACE, ALTER TABLESPACE

CREATE TEXT SEARCH CONFIGURATION

CREATE TEXT SEARCH CONFIGURATION — создать конфигурацию текстового поиска

Синтаксис

```
CREATE TEXT SEARCH CONFIGURATION имя (
    PARSER = имя_анализатора |
    COPY = исходная_конфигурация
)
```

Описание

`CREATE TEXT SEARCH CONFIGURATION` создаёт конфигурацию текстового поиска. Конфигурация текстового поиска определяет анализатор текстового поиска, разделяющий строку на фрагменты, и словари, позволяющие установить, какие именно фрагменты представляют интерес при поиске.

Если задаётся только анализатор, новая конфигурация текстового поиска не будет содержать сопоставления типов фрагментов со словарями и, как следствие, будет игнорировать все слова. Для создания сопоставлений, которые бы сделали конфигурацию полезной, затем нужно будет воспользоваться командой `ALTER TEXT SEARCH CONFIGURATION`. Другой вариант команды позволяет скопировать конфигурацию текстового поиска.

Если указывается имя схемы, конфигурация текстового поиска создаётся в указанной схеме. В противном случае она создаётся в текущей схеме.

Владельцем конфигурации текстового поиска становится пользователь её создавший.

За дополнительными сведениями обратитесь к Главе 12.

Параметры

имя

Имя создаваемой конфигурации текстового поиска, возможно, дополненное схемой.

имя_анализатора

Имя анализатора текстового поиска, который будет использоваться этой конфигурацией.

исходная_конфигурация

Имя существующей конфигурации текстового поиска, которая будет скопирована.

Замечания

Параметры `PARSER` и `COPY` являются взаимоисключающими, так как при копировании существующей конфигурации выбранный в ней анализатор копируется тоже.

Совместимость

Оператор `CREATE TEXT SEARCH CONFIGURATION` отсутствует в стандарте SQL.

См. также

ALTER TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION

CREATE TEXT SEARCH DICTIONARY

CREATE TEXT SEARCH DICTIONARY — создать словарь текстового поиска

Синтаксис

```
CREATE TEXT SEARCH DICTIONARY имя (
    TEMPLATE = шаблон
    [, параметр = значение [, ... ]]
)
```

Описание

`CREATE TEXT SEARCH DICTIONARY` создаёт словарь текстового поиска. Словарь текстового поиска определяет способ выделения слов, представляющих или не представляющих интерес для поиска. Работа словаря зависит от шаблона текстового поиска, в котором определяются функции, собственно выполняющие действия. Обычно словарь задаёт некоторые параметры, управляющие частным поведением функций шаблона.

Если указывается имя схемы, словарь текстового поиска создаётся в указанной схеме. В противном случае он создаётся в текущей схеме.

Владельцем словаря текстового поиска становится пользователь его создавший.

За дополнительными сведениями обратитесь к Главе 12.

Параметры

имя

Имя создаваемого словаря текстового поиска, возможно, дополненное схемой.

шаблон

Имя шаблона текстового поиска, который будет определять общее поведение этого словаря.

параметр

Имя параметра шаблона, устанавливаемого для данного словаря.

значение

Значение для параметра, связанного с шаблоном. Если это не простой идентификатор или число, его следует заключить в кавычки (при желании его можно заключать в кавычки всегда).

Параметры могут перечисляться в любом порядке.

Примеры

Команда в следующем примере создаёт словарь на базе Snowball с нестандартным списком стоп-слов.

```
CREATE TEXT SEARCH DICTIONARY my_russian (
    template = snowball,
    language = russian,
    stopwords = myrussian
);
```

Совместимость

Оператор `CREATE TEXT SEARCH DICTIONARY` отсутствует в стандарте SQL.

См. также

ALTER TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY

CREATE TEXT SEARCH PARSER

CREATE TEXT SEARCH PARSER — создать анализатор текстового поиска

Синтаксис

```
CREATE TEXT SEARCH PARSER имя (
    START = функция_начала ,
    GETTOKEN = функция_выдачи_фрагмента ,
    END = функция_окончания ,
    LEXTYPES = функция_лекс_типов
    [, HEADLINE = функция_выдержек ]
)
```

Описание

`CREATE TEXT SEARCH PARSER` создаёт анализатор текстового поиска. Анализатор текстового поиска определяет способ разделения текстовой строки на фрагменты и назначения типов (категорий) этим фрагментам. Анализатор не очень полезен сам по себе, для осуществления поиска он должен быть подключён к конфигурации текстового поиска вместе с определёнными словарями.

Если указывается имя схемы, словарь текстового поиска создаётся в указанной схеме. В противном случае он создаётся в текущей схеме.

Выполнить `CREATE TEXT SEARCH PARSER` может только суперпользователь. (Это ограничение введено потому, что ошибочное определение анализатора текстового поиска может вызвать нарушения или даже сбой в работе сервера.)

За дополнительными сведениями обратитесь к Главе 12.

Параметры

имя

Имя создаваемого анализатора текстового поиска, возможно, дополненное схемой.

функция_начала

Имя функции, вызываемой в начале обработки.

функция_выдачи_фрагмента

Имя функции, выдающей следующий фрагмент.

функция_окончания

Имя функции, вызываемой по окончании обработки.

функция_лекс_типов

Имя функции перечисления лексических типов (эта функция выдаёт информацию о множестве типов фрагментов, выделяемых анализатором).

функция_выдержек

Имя функции извлечения выдержек (эта функция выделяет краткое содержание для набора фрагментов).

Имена функций могут быть дополнены именем схемы, если требуется. Типы аргументов не указываются, так как список аргументов для всех типов функций предопределён. Обязательными являются все функции, кроме функции выдержек.

Аргументы могут перечисляться в любом порядке, не только в том, что показан выше.

Совместимость

Оператор `CREATE TEXT SEARCH PARSER` отсутствует в стандарте SQL.

См. также

ALTER TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER

CREATE TEXT SEARCH TEMPLATE

CREATE TEXT SEARCH TEMPLATE — создать шаблон текстового поиска

Синтаксис

```
CREATE TEXT SEARCH TEMPLATE имя (
    [ INIT = функция_инициализации , ]
    LEXIZE = функция_выделения_лексем
)
```

Описание

`CREATE TEXT SEARCH TEMPLATE` создаёт новый шаблон текстового поиска. Шаблоны текстового поиска определяют функции для реализации словарей текстового поиска. Шаблон сам по себе бесполезен, но его экземпляр нужно создать для словаря. Словарь обычно определяет параметры, передаваемые функциям шаблона.

Если указывается имя схемы, шаблон текстового поиска создаётся в указанной схеме. В противном случае он создаётся в текущей схеме.

Выполнить `CREATE TEXT SEARCH TEMPLATE` может только суперпользователь. Это ограничение введено потому, что ошибочное определение шаблона текстового поиска может вызвать нарушения или даже сбой в работе сервера. Смысл отделения шаблонов от словарей в том, что шаблон покрывает «небезопасные» аспекты определения словаря. Параметры, которые можно задать при определении словаря, не могут причинить вред, поэтому создавать словари разрешено и непривилегированным пользователям.

За дополнительными сведениями обратитесь к Главе 12.

Параметры

имя

Имя создаваемого шаблона текстового поиска, возможно, дополненное схемой.

функция_инициализации

Имя функции инициализации для шаблона.

функция_выделения_лексем

Имя функции выделения лексем.

Имена функций могут быть дополнены именем схемы, если требуется. Типы аргументов не указываются, так как список аргументов для всех типов функций предопределён. Функция выделения лексем является обязательной, а функция инициализации может отсутствовать.

Аргументы могут перечисляться в любом порядке, не только в том, что показан выше.

Совместимость

Оператор `CREATE TEXT SEARCH TEMPLATE` отсутствует в стандарте SQL.

См. также

ALTER TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE

CREATE TRANSFORM

CREATE TRANSFORM — создать трансформацию

Синтаксис

```
CREATE [ OR REPLACE ] TRANSFORM FOR имя_типа LANGUAGE имя_языка (
    FROM SQL WITH FUNCTION имя_функции_из_sql [ (тип_аргумента [, ...]) ],
    TO SQL WITH FUNCTION имя_функции_в_sql [ (тип_аргумента [, ...]) ]
);
```

Описание

`CREATE TRANSFORM` определяет новую трансформацию. `CREATE OR REPLACE TRANSFORM` либо создаёт трансформацию, либо заменяет существующую.

Трансформация определяет, как преобразовать тип данных для процедурного языка. Например, если написать на языке PL/Python функцию, использующую тип `hstore`, PL/Python заведомо не знает, как должны представляться значения `hstore` в среде Python. Обычно реализации языка нисходят к текстовому представлению, но это может быть неудобно, когда более уместен был бы, например, ассоциативный массив или список.

Трансформация определяет две функции:

- Функция «из SQL» преобразует тип из среды SQL в среду языка. Эта функция будет вызываться для аргументов функции, написанной на этом языке.
- Функция «в SQL» преобразует тип из среды языка в среду SQL. Эта функция будет вызываться для значения, возвращаемого из функции на этом языке.

Предоставлять обе эти функции не требуется, можно ограничиться одной. Если одна из них не указана, при необходимости выбирается поведение, принятое для языка по умолчанию. (Чтобы полностью перекрыть путь трансформации в одну сторону, можно написать функцию, которая будет всегда выдавать ошибку.)

Чтобы создать трансформацию, необходимо быть владельцем и иметь право `USAGE` для типа, иметь право `USAGE` для языка, а также быть владельцем и иметь право `EXECUTE` для функций из-SQL и в-SQL, если они задаются.

Параметры

имя_типа

Имя типа данных, для которого предназначена трансформация.

имя_языка

Имя языка, для которого предназначена трансформация.

имя_функции_из_sql[(*тип_аргумента* [, ...])]

Имя функции для преобразования типа из среды SQL в среду языка. Она должна принимать один аргумент типа `internal` и возвращать тип `internal`. Фактический аргумент будет иметь тип, заданный для трансформации, и сама функция должна рассчитывать на это. (Но на уровне SQL нельзя объявить функцию, возвращающую тип `internal`, если она не принимает минимум один аргумент типа `internal`.) Фактически возвращаемое значение будет определяться реализацией языка. Если список аргументов отсутствует, имя функции должно быть уникальным в её схеме.

имя_функции_в_sql[(*тип_аргумента* [, ...])]

Имя функции для преобразования типа из среды языка в среду SQL. Она должна принимать один аргумент типа `internal` и возвращать тип, для которого создаётся трансформация.

Фактическое значение аргумента будет определяться реализацией языка. Если список аргументов отсутствует, имя функции должно быть уникальным в её схеме.

Замечания

Для удаления трансформаций применяется DROP TRANSFORM.

Примеры

Чтобы создать трансформацию для типа `hstore` и языка `plpythonu`, сначала нужно создать тип и язык:

```
CREATE TYPE hstore ...;

CREATE LANGUAGE plpythonu ...;
```

Затем создайте необходимые функции:

```
CREATE FUNCTION hstore_to_plpython(val internal) RETURNS internal
LANGUAGE C STRICT IMMUTABLE
AS ...;

CREATE FUNCTION plpython_to_hstore(val internal) RETURNS hstore
LANGUAGE C STRICT IMMUTABLE
AS ...;
```

И наконец, создайте трансформацию, соединяющую всё это вместе:

```
CREATE TRANSFORM FOR hstore LANGUAGE plpythonu (
    FROM SQL WITH FUNCTION hstore_to_plpython(internal),
    TO SQL WITH FUNCTION plpython_to_hstore(internal)
);
```

На практике эти команды помещаются в расширение.

В разделе `contrib` представлено несколько расширений, в которых определены трансформации, что может послужить практическим примером реализации.

Совместимость

Первая форма `CREATE TRANSFORM` является расширением Postgres Pro. В стандарте SQL есть команда `CREATE TRANSFORM`, но её предназначение — преобразовывать типы для языков на стороне клиента. Этот вариант использования не поддерживается Postgres Pro.

См. также

CREATE FUNCTION, CREATE LANGUAGE, CREATE TYPE, DROP TRANSFORM

CREATE TRIGGER

CREATE TRIGGER — создать триггер

Синтаксис

```
CREATE [ CONSTRAINT ] TRIGGER имя { BEFORE | AFTER | INSTEAD OF } { событие
 [ OR ... ] }
    ON имя_таблицы
    [ FROM ссылающаяся_таблица ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
    [ REFERENCING { { OLD | NEW } TABLE [ AS ] имя_переходного_отношения } [ ... ] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( условие ) ]
    EXECUTE PROCEDURE имя_функции ( аргументы )

Здесь допускается событие:

    INSERT
    UPDATE [ OF имя_столбца [, ... ] ]
    DELETE
    TRUNCATE
```

Описание

`CREATE TRIGGER` создаёт новый триггер. Триггер будет связан с указанной таблицей, представлением или сторонней таблицей и будет выполнять заданную функцию *имя_функции* при определённых операциях с этой таблицей.

Триггер можно настроить так, чтобы он срабатывал до операции со строкой (до проверки ограничений и попытки выполнить `INSERT`, `UPDATE` или `DELETE`) или после её завершения (после проверки ограничений и выполнения `INSERT`, `UPDATE` или `DELETE`), либо вместо операции (при добавлении, изменении и удалении строк в представлении). Если триггер срабатывает до или вместо события, он может пропустить операцию с текущей строкой, либо изменить добавляемую строку (только для операций `INSERT` и `UPDATE`). Если триггер срабатывает после события, он «видит» все изменения, включая результат действия других триггеров.

Триггер с пометкой `FOR EACH ROW` вызывается один раз для каждой строки, изменяемой в процессе операции. Например, операция `DELETE`, удаляющая 10 строк, приведёт к срабатыванию всех триггеров `ON DELETE` в целевом отношении 10 раз подряд, по одному разу для каждой удаляемой строки. Триггер с пометкой `FOR EACH STATEMENT`, напротив, вызывается только один раз для конкретной операции, вне зависимости от того, как много строк она изменила (в частности, при выполнении операции, изменяющей ноль строк, всё равно будут вызваны все триггеры `FOR EACH STATEMENT`).

Триггеры, срабатывающие в режиме `INSTEAD OF`, должны быть помечены `FOR EACH ROW` и могут быть определены только для представлений. Триггеры `BEFORE` и `AFTER` для представлений должны быть помечены `FOR EACH STATEMENT`.

Кроме того, триггеры можно определить и для команды `TRUNCATE`, но только типа `FOR EACH STATEMENT`.

В следующей таблице перечисляются типы триггеров, которые могут использоваться для таблиц, представлений и сторонних таблиц:

Когда	Событие	На уровне строк	На уровне оператора
BEFORE	INSERT/UPDATE/DELETE	Таблицы и сторонние таблицы	Таблицы, представления и сторонние таблицы
	TRUNCATE	—	Таблицы
AFTER	INSERT/UPDATE/DELETE	Таблицы и сторонние таблицы	Таблицы, представления и сторонние таблицы
	TRUNCATE	—	Таблицы
INSTEAD OF	INSERT/UPDATE/DELETE	Представления	—
	TRUNCATE	—	—

Кроме того, в определении триггера можно указать логическое условие WHEN, которое определит, вызывать триггер или нет. В триггерах на уровне строк условия WHEN могут проверять старые и/или новые значения столбцов в строке. Триггеры на уровне оператора так же могут содержать условие WHEN, хотя для них это не столь полезно, так как в этом условии нельзя ссылаться на какие-либо значения в таблице.

Если для одного события определено несколько триггеров одного типа, они будут срабатывать в алфавитном порядке их имён.

Когда указывается параметр CONSTRAINT, эта команда создаёт *триггер ограничения*. Он подобен обычным триггерам, но отличается тем, что время его срабатывания можно изменить командой SET CONSTRAINTS. Триггеры ограничений должны быть триггерами типа AFTER ROW для обычных (не сторонних) таблиц. Они могут срабатывать либо в конце оператора, вызвавшего целевое событие, либо в конце содержащей его транзакции; в последнем случае они называются *отложенными*. Срабатывание ожидающего отложенного триггера можно вызвать немедленно, воспользовавшись командой SET CONSTRAINTS. Предполагается, что триггеры ограничений будут генерировать исключения при нарушении ограничений.

Когда указывается REFERENCING, для триггера собираются *переходные отношения*, представляющие собой множества строк, включающие все строки, которые были добавлены, удалены или изменены текущим оператором SQL. Это позволяет триггеру наблюдать общую картину того, что сделал оператор, а не только одну строку за другой. Это указание допускается только для триггера AFTER, не являющегося триггером ограничения; кроме того, если это триггер для UPDATE, у него должен отсутствовать список *имён_столбцов*. Указание OLD TABLE может быть задано только один раз и только для триггера, который может срабатывать при UPDATE или DELETE; оно создаёт переходное отношение, содержащее *образы-до-изменения* всех строк, модифицированных или удалённых оператором. Указание NEW TABLE, подобным образом, может быть задано только единожды и только для триггера, который может срабатывать для UPDATE или INSERT; оно создаёт переходное отношение, содержащее *образы-после-изменения* всех строк, модифицированных или добавленных оператором.

SELECT не изменяет никакие строки, поэтому создавать триггеры для SELECT нельзя. Для решения задач, в которых требуются подобные триггеры, могут подойти правила или представления.

За дополнительными сведениями о триггерах обратитесь к Главе 36.

Параметры

имя

Имя, назначаемое новому триггеру. Это имя должно отличаться от имени любого другого триггера в этой же таблице. Имя не может быть дополнено схемой — триггер наследует схему от своей таблицы. Для триггеров ограничений это имя также используется, когда требуется скорректировать поведение триггера с помощью команды SET CONSTRAINTS.

BEFORE
AFTER
INSTEAD OF

Определяет, будет ли заданная функция вызываться до, после или вместо события. Для триггера ограничения можно указать только AFTER.

событие

Принимает одно из значений: INSERT, UPDATE, DELETE или TRUNCATE; этот параметр определяет событие, при котором будет срабатывать триггер. Несколько событий можно указать, добавив между ними слово OR, если только не запрашиваются переходные отношения.

Для событий UPDATE можно указать список столбцов, используя такую запись:

UPDATE OF *имя_столбца1* [, *имя_столбца2* ...]

Такой триггер сработает, только если в указанном в целевой команде UPDATE списке столбцов окажется минимум один из перечисленных.

Для событий INSTEAD OF UPDATE указание списка столбцов не допускается. Список столбцов также нельзя задать, когда запрашиваются переходные отношения.

имя_таблицы

Имя (возможно, дополненное схемой) таблицы, представления или сторонней таблицы, для которых предназначен триггер.

ссылающаяся_таблица

Имя (возможно, дополненное схемой) другой таблицы, на которую ссылается ограничение. Оно используется для ограничений внешнего ключа и не рекомендуется для обычного применения. Это указание допускается только для триггеров ограничений.

DEFERRABLE
NOT DEFERRABLE
INITIALLY IMMEDIATE
INITIALLY DEFERRED

Время срабатывания триггера по умолчанию. Подробнее возможные варианты описаны в документации CREATE TABLE. Это указание допускается только для триггеров ограничений.

REFERENCING

Это ключевое слово непосредственно предшествует объявлению одного или двух имён, по которым можно будет обращаться к переходным отношениями, образуемым при выполнении целевого оператора.

OLD TABLE
NEW TABLE

Это предложение указывает, будет ли следующее имя относиться к переходному отношению с образом-до-изменения или к переходному отношению с образом-после-изменения.

имя_переходного_отношения

Имя (неполное, без схемы), которое будет использоваться в триггере для обращения к этому переходному отношению.

FOR EACH ROW
FOR EACH STATEMENT

Определяет, будет ли процедура триггера срабатывать один раз для каждой строки, либо для SQL-оператора. Если не указано ничего, подразумевается FOR EACH STATEMENT (для оператора). Для триггеров ограничений можно указать только FOR EACH ROW.

условие

Логическое выражение, определяющее, будет ли выполняться функция триггера. Если для триггера задано указание WHEN, функция будет вызываться, только когда *условие* возвращает true. В триггерах FOR EACH ROW условие WHEN может ссылаться на значения столбца в старой и/или новой строке, в виде OLD.*имя_столбца* и NEW.*имя_столбца*, соответственно. Разумеется, триггеры INSERT не могут ссылаться на OLD, а триггеры DELETE не могут ссылаться на NEW.

Триггеры INSTEAD OF не поддерживают условия WHEN.

В настоящее время выражения WHEN не могут содержать подзапросы.

Учтите, что для триггеров ограничений вычисление условия WHEN не откладывается, а выполняется немедленно после операции, изменяющей строки. Если результат условия — ложь, сам триггер не откладывается для последующего выполнения.

имя_функции

Заданная пользователем функция, объявленная как функция без аргументов и возвращающая тип trigger, которая будет вызываться при срабатывании триггера.

аргументы

Необязательный список аргументов через запятую, которые будут переданы функции при срабатывании триггера. В качестве аргументов функции передаются строковые константы. И хотя в этом списке можно записать и простые имена или числовые константы, они тоже будут преобразованы в строки. Порядок обращения к таким аргументам в функции триггера может отличаться от обычных аргументов, поэтому его следует уточнить в описании языка реализации этой функции.

Замечания

Чтобы создать триггер, пользователь должен иметь право TRIGGER для этой таблицы. Также пользователь должен иметь право EXECUTE для триггерной функции.

Для удаления триггера применяется команда DROP TRIGGER.

Триггер для избранных столбцов (определённый с помощью UPDATE OF *имя_столбца*) будет срабатывать, когда его столбцы перечислены в качестве целевых в списке SET команды UPDATE. Изменения, вносимые в строки триггерами BEFORE UPDATE, при этом не учитываются, поэтому значения столбцов можно изменить так, что триггер не сработает. И наоборот, при выполнении команды UPDATE ... SET x = x ... триггер для столбца x сработает, хотя значение столбца не меняется.

Некоторые общие задачи можно решить с применением встроенных триггерных функций, обойдясь без написания собственного кода; см. Раздел 9.27.

В триггере BEFORE условие WHEN вычисляется непосредственно перед возможным вызовом функции, поэтому проверка WHEN существенно не отличается от проверки того же условия в начале функции триггера. В частности, учтите, что строка NEW, которую видит ограничение, содержит текущие значения, возможно изменённые предыдущими триггерами. Кроме того, в триггере BEFORE условие WHEN не может проверять системные столбцы в строке NEW (например, oid), так как они ещё не установлены.

В триггере AFTER условие WHEN проверяется сразу после изменения строки, и если оно выполняется, событие запоминается, чтобы вызвать триггер в конце оператора. Если же для триггера AFTER условие WHEN не выполняется, нет необходимости запоминать событие для последующей обработки или заново перечитывать строку в конце оператора. Это приводит к значительному ускорению операторов, изменяющих множество строк, когда триггер должен срабатывать только для некоторых из них.

В некоторых случаях одна команда SQL может вызывать сразу нескольких видов триггеров. Например, INSERT с предложением ON CONFLICT DO UPDATE может выполнять операции как добавления, так и изменения, так что она при необходимости будет вызывать триггеры обоих видов. При этом переходные отношения, предоставляемые триггерам, будут разными в зависимости от типа события; то есть триггер INSERT будет видеть только добавленные строки, а триггер UPDATE — только изменённые.

Изменения или удаления строк, вызванные действиями по обеспечению целостности внешнего ключа, например, ON UPDATE CASCADE или ON DELETE SET NULL, считаются частью SQL-команды, вызвавшей эти действия (заметьте, что такие действия не могут быть отложенными). В затрагиваемой таблице будут вызваны соответствующие триггеры, и таким образом появляется возможность вызова триггеров для SQL-команды, не соответствующей непосредственно их типу. В простых ситуациях триггеры, запрашивающие переходные отношения, будут видеть все изменения, произведённые в их таблице одной исходной командой SQL, в виде одного переходного отношения. Однако возможны случаи, в которых присутствие триггера AFTER ROW, запрашивающего переходные отношения, приведёт к тому, что операции для обеспечения целостности внешнего ключа, вызванные одной SQL-командой, будут разделены на несколько этапов, и на каждом будут свои переходные отношения. В таких случаях все существующие триггеры уровня оператора будут срабатывать единожды при создании переходного отношения, что гарантирует, что эти триггеры будут видеть каждую обрабатываемую строку в переходном отношении один и только один раз.

Триггеры уровня операторов для представления срабатывают, только если операция с представлением обрабатывается триггером уровня строк INSTEAD OF. Если операция обрабатывается правилом INSTEAD, то вместо исходного оператора, обращающегося к представлению, выполняются те операторы, что генерирует правило, поэтому вызываться будут триггеры, связанные с таблицами, к которым обращаются эти заменяющие операторы. Аналогично, для автоматически изменяемого представления выполнение операции сводится к переписыванию оператора в виде операции с базовой таблицей представления, так что срабатывать будут триггеры уровня операторов для базовой таблицы.

При изменении данных в секционированной таблице или таблице с потомками срабатывают триггеры уровня оператора, связанные с явно задействованной таблицей, но не триггеры уровня оператора для её секций или дочерних таблиц. Триггеры уровня строк, напротив, срабатывают для строк в затрагиваемых секциях или дочерних таблицах, даже если они явно не присутствуют в запросе. Если триггер уровня оператора был определён с переходными отношениями, названными в указании REFERENCING, то в них будут видны образы строк из всех затронутых секций или дочерних таблиц. В случае с потомками в иерархии наследования образы строк будут содержать только столбцы, присутствующие в таблице, с которой связан триггер. В настоящее время триггеры уровня строк с переходными отношениями нельзя определить для секций или дочерних таблиц в иерархии наследования.

В PostgreSQL до версии 7.3 обязательно требовалось объявлять триггерные функции, как возвращающие фиктивный тип opaque, а не trigger. Для поддержки загрузки старых файлов экспорта БД, команда CREATE TRIGGER принимает функции с объявленным типом результата opaque, но при этом выдаётся предупреждение и тип результата меняется на trigger.

Примеры

Выполнение функции check_account_update перед любым изменением строк в таблице accounts:

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    EXECUTE PROCEDURE check_account_update();
```

То же самое, но функция триггера будет выполняться, только если столбец balance присутствует в списке целевых столбцов команды UPDATE:

```
CREATE TRIGGER check_update
```

```
    BEFORE UPDATE OF balance ON accounts
    FOR EACH ROW
    EXECUTE PROCEDURE check_account_update();
```

В этом примере функция будет выполняться, если значение столбца `balance` в действительности изменилось:

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
    EXECUTE PROCEDURE check_account_update();
```

Вызов функции, ведущей журнал изменений в `accounts`, но только если что-то изменилось:

```
CREATE TRIGGER log_update
    AFTER UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.* IS DISTINCT FROM NEW.*)
    EXECUTE PROCEDURE log_account_update();
```

Выполнение для каждой строки функции `view_insert_row`, которая будет вставлять строки в нижележащие таблицы представления:

```
CREATE TRIGGER view_insert
    INSTEAD OF INSERT ON my_view
    FOR EACH ROW
    EXECUTE PROCEDURE view_insert_row();
```

Выполнение функции `check_transfer_balances_to_zero` для каждого оператора, проверяющей, что строки `transfer` в совокупности дают нулевой баланс:

```
CREATE TRIGGER transfer_insert
    AFTER INSERT ON transfer
    REFERENCING NEW TABLE AS inserted
    FOR EACH STATEMENT
    EXECUTE PROCEDURE check_transfer_balances_to_zero();
```

Выполнение функции `check_matching_pairs` для каждой строки, проверяющей, что соответствующие пары пунктов изменены синхронно (одним оператором):

```
CREATE TRIGGER paired_items_update
    AFTER UPDATE ON paired_items
    REFERENCING NEW TABLE AS newtab OLD TABLE AS oldtab
    FOR EACH ROW
    EXECUTE PROCEDURE check_matching_pairs();
```

В Разделе 36.4 приведён полный пример функции триггера, написанной на C.

Совместимость

Оператор `CREATE TRIGGER` в Postgres Pro реализует подмножество возможностей, описанных в стандарте SQL. В настоящее время в нём отсутствует следующая функциональность:

- Тогда как имена переходных таблиц для триггеров `AFTER` задаются предложением `REFERENCING` стандартным образом, переменные строк, применяемые в триггерах `FOR EACH ROW` нельзя объявлять в предложении `REFERENCING`. Порядок обращения к таким строкам зависит от языка, на котором написана триггерная функция, но для каждого языка он вполне определённый. Некоторые языки по сути действуют так, как будто в команде присутствует предложение `REFERENCING` с указанием `OLD ROW AS OLD NEW ROW AS NEW`.
- Стандарт позволяет использовать переходные таблицы с триггерами `UPDATE`, ограничивающими набор отслеживаемых столбцов, но тогда и набор строк, видимых в переходных таблицах, должен зависеть от списка целевых столбцов триггера. В настоящее время такое поведение в Postgres Pro не реализовано.

- Postgres Pro позволяет задать в качестве действия триггера только функцию, определённую пользователем. Стандарт допускает также выполнение ряда других команд SQL, например, `CREATE TABLE`. Однако это ограничение несложно преодолеть, создав пользовательскую функцию, выполняющую требуемые команды.

В стандарте SQL определено, что несколько триггеров должны срабатывать по порядку создания. Postgres Pro упорядочивает их по именам, так как это было признано более удобным.

В стандарте SQL определено, что триггеры `BEFORE DELETE` при каскадном удалении срабатывают *после* завершения каскадного `DELETE`. В Postgres Pro триггеры `BEFORE DELETE` всегда срабатывают перед операцией удаления, даже если она каскадная. Это поведение выбрано как более логичное. Ещё одно отклонение от стандарта проявляется, когда триггеры `BEFORE`, срабатывающие в результате ссылочной операции, изменяют строки или не дают выполнить изменение. Это может привести к нарушению ограничений или сохранению данных, не соблюдающих ссылочную целостность.

Возможность задать несколько действий для одного триггера с помощью ключевого слова `OR` — реализованное в Postgres Pro расширение стандарта SQL.

Возможность вызывать триггеры для `TRUNCATE` — реализованное в Postgres Pro расширение стандарта SQL, как и возможность определять триггеры на уровне оператора для представлений.

`CREATE CONSTRAINT TRIGGER` — реализованное в Postgres Pro расширение стандарта SQL.

См. также

ALTER TRIGGER, DROP TRIGGER, CREATE FUNCTION, SET CONSTRAINTS

CREATE TYPE

CREATE TYPE — создать новый тип данных

Синтаксис

```
CREATE TYPE имя AS
    ( [ имя_атрибута тип_данных [ COLLATE правило_сортировки ] [, ... ] ] )

CREATE TYPE имя AS ENUM
    ( [ 'метка' [, ... ] ] )

CREATE TYPE имя AS RANGE (
    SUBTYPE = подтип
    [ , SUBTYPE_OPCLASS = класс_оператора_подтипа ]
    [ , COLLATION = правило_сортировки ]
    [ , CANONICAL = каноническая_функция ]
    [ , SUBTYPE_DIFF = функция_разницы_подтипа ]
)

CREATE TYPE имя (
    INPUT = функция_ввода,
    OUTPUT = функция_вывода
    [ , RECEIVE = функция_получения ]
    [ , SEND = функция_отправки ]
    [ , TYPMOD_IN = функция_ввода_модификатора_типа ]
    [ , TYPMOD_OUT = функция_вывода_модификатора_типа ]
    [ , ANALYZE = функция_анализа ]
    [ , INTERNALLENGTH = { внутр_длина | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = выравнивание ]
    [ , STORAGE = хранение ]
    [ , LIKE = тип_образец ]
    [ , CATEGORY = категория ]
    [ , PREFERRED = предпочитаемый ]
    [ , DEFAULT = по_умолчанию ]
    [ , ELEMENT = элемент ]
    [ , DELIMITER = разделитель ]
    [ , COLLATABLE = сортируемый ]
)

CREATE TYPE имя
```

Описание

`CREATE TYPE` регистрирует новый тип данных для использования в текущей базе данных. Владельцем типа становится создавший его пользователь.

Если указано имя схемы, тип создаётся в указанной схеме. В противном случае он создаётся в текущей схеме. Имя типа должно отличаться от имён любых других существующих типов или доменов в той же схеме. (А так как с таблицами связываются типы данных, имя типа должно также отличаться и от имён существующих таблиц в этой схеме.)

Команда `CREATE TYPE` имеет пять форм, показанных выше в сводке синтаксиса. Они создают соответственно *составной тип*, *перечисление*, *диапазон*, *базовый тип* или *тип-пустышку*. Первые четыре эти типа рассматриваются по порядку ниже. Тип-пустышка представляет собой просто заготовку для типа, который будет определён позже; он создаётся командой `CREATE TYPE` с

одним именем, без параметров. Типы-пустышки необходимы для определения прямых ссылок при создании базовых типов и типов-диапазонов, как описывается в соответствующих разделах.

Составные типы

Первая форма `CREATE TYPE` создаёт составной тип. Составной тип задаётся списком имён и типами данных атрибутов. Если тип данных является сортируемым, то для атрибута можно также задать правило сортировки. Составной тип по сути не отличается от типа строки таблицы, но `CREATE TYPE` избавляет от необходимости создавать таблицу, когда всё, что нужно, это создать тип. Отдельный составной тип может быть полезен, например, для передачи аргументов или результатов функции.

Чтобы создать составной тип, необходимо иметь право `USAGE` для типов всех его атрибутов.

Типы перечислений

Вторая форма `CREATE TYPE` создаёт тип-перечисление (такие типы описываются в Разделе 8.7). Перечисления принимают список меток в кавычках. Максимальная длина каждой метки — `NAMEDATALEN` байт (64 байта в стандартной сборке Postgres Pro). (Также возможно создать перечисляемый тип без меток, но этот тип нельзя будет использовать для хранения значений, пока командой ALTER TYPE не будет добавлена хотя бы одна метка.)

Диапазонные типы

Третья форма `CREATE TYPE` создаёт тип-диапазон (такие типы описываются в Разделе 8.17).

Задаваемый для диапазона *`подтип`* может быть любым типом со связанным классом операторов B-дерева (что позволяет определить порядок значений в диапазоне). Обычно порядок элементов определяет класс операторов B-дерева по умолчанию, но его можно изменить, задав имя другого класса в параметре *`класс_операторов_подтипа`*. Если подтип поддерживает сортировку и требуется, чтобы значения упорядочивались с нестандартным правилом сортировки, его имя можно задать в параметре *`правило_сортировки`*.

Необязательная *`каноническая_функция`* должна принимать один аргумент определяемого типа диапазона и возвращать значение того же типа. Это используется для преобразования значений диапазона в каноническую форму, когда это уместно. За дополнительными сведениями обратитесь к Подразделу 8.17.8. Создаётся *`каноническая_функция`* несколько нетривиально, так как она должна быть уже определена, прежде чем можно будет объявить тип-диапазон. Для этого нужно сначала создать тип-пустышку, который будет заготовкой типа, не имеющей никаких свойств, кроме имени и владельца. Это можно сделать, выполнив команду `CREATE TYPE` *`имя`* без дополнительных параметров. Затем можно объявить функцию, для которой тип-пустышка будет типом аргумента и результата, и, наконец, объявить тип-диапазон с тем же именем. При этом тип-пустышка автоматически заменится полноценным типом-диапазоном.

Необязательная *`функция_разницы_подтипа`* должна принимать в аргументах два значения типа *`подтип`* и возвращать значение `double precision`, представляющее разницу между двумя данными значениями. Хотя эту функцию можно не использовать, она позволяет кардинально увеличить эффективность индексов GiST для столбцов с типом-диапазоном. За дополнительными сведениями обратитесь к Подразделу 8.17.8.

Базовые типы

Четвёртая форма `CREATE TYPE` создаёт новый базовый тип (скалярный тип). Чтобы создать новый базовый тип, нужно быть суперпользователем. (Это ограничение введено потому, что ошибочное определение типа может вызвать нарушения или даже сбой в работе сервера.)

Эти параметры могут перечисляться в любом порядке, не только в показанном выше, и большинство из них необязательные. Прежде чем создавать тип, необходимо зарегистрировать две или более функций (с помощью `CREATE FUNCTION`). Обязательными являются функции *`функция_ввода`* и *`функция_вывода`*, тогда как *`функция_получения`*, *`функция_отправки`*, *`функция_модификатора_типа`*, *`функция_вывода_модификатора_типа`* и *`функция_анализа`* могут отсутствовать. Обычно эти функции разрабатываются на C или другом низкоуровневом языке.

Функция_ввода преобразует внешнее текстовое представление типа во внутреннее, с которым работают операторы и функции, определённые для этого типа. *Функция_вывода* выполняет обратное преобразование. Функцию ввода можно объявить как принимающую один аргумент типа cstring, либо как принимающую три аргумента типов cstring, oid и integer. В первом аргументе передаётся вводимый текст в виде строки в стиле C, во втором аргументе — собственный OID типа (кроме типов массивов, для которых передаётся OID типа элемента), а в третьем — модификатор_типа для целевого столбца, если он определён (или -1 в противном случае). Функция ввода должна возвращать значение нового типа данных. Обычно функция ввода должна быть строгой (STRICT); если это не так, при получении на вход значения NULL она будет вызываться с первым параметром NULL. Функция может в этом случае сама вернуть NULL или вызвать ошибку. (Это полезно в основном для поддержки функций ввода доменных типов, которые не должны принимать данные NULL.) Функция вывода должна принимать один аргумент нового типа данных, а возвращать она должна cstring. Для значений NULL функции вывода не вызываются.

Необязательная *функция_получения* преобразует двоичное внешнее представление типа во внутреннее представление. Если эта функция отсутствует, новый тип не сможет участвовать в двоичном вводе. Двоичное представление следует выбирать таким, чтобы оно легко переводилось во внутреннюю форму и при этом было переносимым до разумной степени. (Например, для стандартных целочисленных типов данных во внешнем двоичном представлении выбран сетевой порядок байтов, тогда как внутреннее представление определяется порядком байтов в процессоре.) Функция получения должна выполнить проверку вводимого значения на допустимость. Функция получения может быть объявлена как принимающая один аргумент типа internal, либо как принимающая три аргумента типов internal, oid и integer. В первом аргументе передаётся указатель на буфер StringInfo, содержащий полученную байтовую строку, а дополнительные аргументы такие же, как и для функции ввода текста. Функция получения должна возвращать значение нового типа данных. Обычно функция получения должна быть строгой (STRICT); если это не так, при получении на вход значения NULL, она будет вызываться с первым параметром NULL. Функция может в этом случае сама вернуть NULL или вызывать ошибку. (Это полезно в основном для поддержки функций получения доменных типов, которые не должны принимать значения NULL.) Подобным образом, необязательная *функция_отправки* преобразует данные из внутреннего во внешнее двоичное представление. Если эта функция не определена, новый тип не может участвовать в двоичном выводе. Функция отправки должна принимать один аргумент нового типа данных, а возвращать она должна bytea. Для значений NULL функции отправки не вызываются.

Здесь у вас может возникнуть вопрос, как функции ввода и вывода могут быть объявлены принимающими или возвращающими значения нового типа, если они должны быть созданы до объявления нового типа. Ответ довольно прост: сначала нужно создать *тип-пустышку*, который будет заготовкой типа, не имеющей никаких свойств, кроме имени и владельца. Это можно сделать, выполнив команду CREATE TYPE *имя* без дополнительных параметров. Затем можно будет определить функции ввода/вывода на C, ссылающиеся на этот тип. И наконец, команда CREATE TYPE с полным определением заменит тип-пустышку окончательным и полноценным определением, после чего новый тип можно будет использовать как обычно.

Необязательные *функция_ввода_модификатора_типа* и *функция_вывода_модификатора_типа* требуются, только если типы поддерживают модификаторы, или, другими словами, дополнительные ограничения, связываемые с объявлением типа, например char(5) или numeric(30,2). В Postgres Pro типы могут принимать в качестве модификаторов одну или несколько простых констант или идентификаторов. Однако эти данные должны упаковываться в единственное неотрицательное целочисленное значение, которое и будет храниться в системных каталогах. *Функция_ввода_модификатора_типа* получает объявленные модификаторы в виде строки cstring. Она должна проверить значения на допустимость (и вызвать ошибку, если они неверны), а затем выдать неотрицательное значение integer, которое будет сохранено в столбце «typmod». Если для типа не определена *функция_ввода_модификатора_типа*, модификаторы типа приниматься не будут. *Функция_вывода_модификатора_типа* преобразует внутреннее целочисленное значение typmod обратно, в форму, понятную пользователю. Она должна вернуть значение cstring, которое именно в этом виде будет добавлено к имени типа; например, функция для numeric должна вернуть (30,2). *Функция_вывода_модификатора_типа* может быть опущена, в этом случае

сохранённое целочисленное значение typmod по умолчанию будет выводиться просто в виде числа, заключённого в скобки.

Необязательная *функция_анализа* выполняет сбор специфической для этого типа статистики в столбцах с таким типом данных. По умолчанию ANALYZE пытается собрать статистику, используя операторы «равно» и «меньше», если для этого типа определён класс операторов B-дерева по умолчанию. Для нескалярных типов это поведение скорее всего не подойдёт, поэтому его можно переопределить, задав собственную функцию анализа. Эта функция должна принимать единственный аргумент типа internal и возвращать результат boolean. Более глубоко API функций анализа описан в src/include/commands/vacuum.h.

Если особенности внутреннего представления нового типа известны функциям ввода/вывода и другим функциям, созданным специально для работы с этим типом, необходимо определить ряд характеристик внутреннего представления, о которых должен знать Postgres Pro. В первую очередь это *internallength* (внутренняя длина). Если базовый тип данных имеет фиксированную длину, в *internallength* указывается эта длина в виде положительного числа, а если длина переменная, в *internallength* задаётся значение VARIABLE. (Внутри при этом typlen принимает значение -1.) Внутреннее представление всех типов переменной длины должно начинаться с 4-байтового целого, задающего общую длину значения этого типа. (Заметьте, что поле длины часто кодируется, как описано в Разделе 63.2; обращаться к нему напрямую неразумно.)

Необязательный флаг PASSEDBYVALUE указывает, что значения этого типа данных передаются по значению, а не по ссылке. Типы, передаваемые по значению, должны быть фиксированной длины и их внутреннее представление не может быть больше размера типа Datum (4 байта на одних машинах, 8 — на других).

Параметр *выравнивание* определяет, как требуется выравнивать данные этого типа. Допускается выравнивание по границам 1, 2, 4 или 8 байт. Заметьте, что типы переменной длины должны быть выровнены как минимум по границе 4 байт, так как их первым компонентом обязательно должен быть int4.

Параметр *хранение* позволяет выбрать стратегию хранения для типов данных переменной длины. (Для типов с фиксированной длиной поддерживается только вариант plain.) Если выбрана стратегия plain, данные этого типа всегда хранятся внутри, без сжатия. Со стратегией extended система сначала попытается сжать большое значение, а затем выносит его из строки основной таблицы, если оно всё же окажется слишком большим. С external значение может быть вынесено из основной таблицы, но система не будет пытаться сжать его. Стратегия main позволяет сжать данные, но не стремится вынести их из основной таблицы. (Элементы данных с этой стратегией хранения тем не менее могут быть вынесены из основной таблицы, если другого способа уместить их в строке нет, но всё же она отдаёт большее предпочтение основной таблице, по сравнению со стратегиями extended и external.)

Значения параметра *хранение*, отличные от plain, подразумевают, что функции типа данных могут принимать значения в формате *toast*, описанном в Разделе 63.2 и Подразделе 35.11.1. Эти значения просто определяют стратегию хранения TOAST по умолчанию для столбцов отделяемого в TOAST типа данных; пользователи могут выбирать другие стратегии для отдельных столбцов, применяя команду ALTER TABLE SET STORAGE.

Параметр *тип_образец* позволяет задать основные свойства представления типа другим способом: скопировать их из существующего типа. В частности, из указанного типа будут скопированы свойства *internallength*, *passedbyvalue*, *alignment* и *storage*. (Также возможно, хотя обычно это не требуется, переопределить некоторые из этих значений, указав их вместе с предложением LIKE.) Определять представление типа таким образом особенно удобно, когда низкоуровневая реализация нового типа некоторым образом опирается на существующий тип.

Параметры *категория* и *предпочитаемый* позволяют определять, какое неявное приведение будет применяться в неоднозначных ситуациях. Каждый тип данных принадлежит к некоторой категории, обозначаемой одним символом ASCII, при этом он может быть, либо не быть «предпочитаемым» в этой категории. Анализатор запроса по возможности выберет приведение к

предпочитаемому типу (но только среди других типов той же категории), когда это может помочь разрешить имя перегруженной функции или оператора. За дополнительными подробностями обратитесь к Главе 10. Если для типа не определено неявное приведение к какому-либо другому типу или обратное, для этих параметров достаточно оставить значения по умолчанию. Однако если есть группа связанных типов, для которых определены неявные приведения, часто бывает полезно пометить их все как принадлежащие некоторой категории и назначить один или два «наиболее общих» предпочитаемыми в этой категории. Параметр *категория* особенно полезен при добавлении типа, определённого пользователем, в существующую встроенную категорию, например, в категорию числовых или строковых типов. Однако так же возможно создать категории типов, полностью определённые пользователем. В качестве имени такой категории можно выбрать любой ASCII-символ, кроме латинской заглавной буквы.

Если пользователь хочет назначить столбцам с этим типом данных значение по умолчанию, отличное от NULL, он может задать его в этой команде, указав его после ключевого слова DEFAULT. (Такое значение по умолчанию можно переопределить явным предложением DEFAULT, добавленным при создании столбца.)

Чтобы обозначить, что тип является массивом, укажите тип элементов массива, добавив ключевое слово ELEMENT. Например, чтобы определить массив из четырёхбайтовых целых (int4), укажите ELEMENT = int4. Дополнительные сведения о типах массивов приведены ниже.

Параметр *delimiter* позволяет задать разделитель, который будет вставляться между значениями во внешнем представлении массива с элементами этого типа. По умолчанию разделителем является запятая (,). Заметьте, что разделитель связывается с типом элементов массива, а не с типом самого массива.

Если необязательный логический параметр *сортируемый* равен true, определения столбцов и выражения с этим типом могут включать указания о порядке сортировки, в предложении COLLATE. Как именно будут использоваться эти указания, зависит от реализации функций, работающих с этим типом; эти указания не действуют автоматически просто от того, что тип помечен как сортируемый.

Типы массивов

При создании любого нового типа Postgres Pro автоматически создаёт соответствующий тип массива, имя которого он получает, добавляя подчёркивание перед именем типа элементов. Если полученное имя оказывается не короче NAMEDATALEN байт, оно усекается. (Если полученное таким образом имя конфликтует с именем уже существующего типа, процесс повторяется, пока не будет получено уникальное имя.) Этот неявно создаваемый тип массива имеет переменную длину и использует встроенные функции ввода и вывода array_in и array_out. Тип массива отражает любые изменения владельца или схемы связанного типа элемента и удаляется сам при удалении типа элемента.

Вы можете вполне резонно спросить, зачем нужен параметр ELEMENT, если система создаёт правильный тип массива автоматически. Единственный случай, когда параметр ELEMENT может быть полезен, это когда вы создаёте тип фиксированной длины, который внутри оказывается массивом одинаковых элементов, и вы хотите, чтобы к этим элементам можно было обращаться по индексу, помимо того, что вы можете реализовать какие угодно операции с типом в целом. Например, тип point представлен просто как два числа с плавающей точкой, к которым можно обратиться так: point[0] и point[1]. Заметьте, что это работает только с типами фиксированной длины, которые представляют собой в точности последовательность одинаковых полей фиксированной длины. Тип массива переменной длины должен иметь обобщённое внутреннее представление, с которым умеют работать array_in и array_out. По историческим причинам (т. е. это определённо некорректно, но менять уже слишком поздно), индексы в массивах фиксированной длины начинаются с нуля, а не с 1, как в массивах переменной длины.

Параметры

имя

Имя создаваемого типа (возможно, дополненное схемой).

имя_атрибута

Имя атрибута (столбца) составного типа.

тип_данных

Имя существующего типа данных, который станет типом столбца составного типа.

правило_сортировки

Имя существующего правила сортировки, связываемого со столбцом составного типа или с типом-диапазоном.

метка

Строковая константа, представляющая текстовую метку, связанную с отдельным значением типа-перечисления.

подтип

Имя типа элемента, множество значений которого будет представлять тип-диапазон.

класс_оператора_подтипа

Имя класса операторов B-дерева для подтипа.

каноническая_функция

Имя функции канонизации для типа-диапазона.

функция_разницы_подтипа

Имя функции разницы для значений подтипа.

функция_ввода

Имя функции, преобразующей данные из внешнего текстового представления типа во внутреннюю форму.

функция_вывода

Имя функции, преобразующей данные из внутренней формы во внешнее текстовое представление типа.

функция_получения

Имя функции, преобразующей данные из внешнего двоичного представления типа во внутреннюю форму.

функция_отправки

Имя функции, преобразующей данные из внутренней формы во внешнее двоичное представление типа.

функция_ввода_модификатора_типа

Имя функции, преобразующей массив модификаторов типа во внутреннюю форму.

функция_вывода_модификатора_типа

Имя функции, преобразующей внутреннюю форму модификаторов типа во внешнее текстовое представление.

функция_анализа

Имя функции, производящей статистический анализ типа данных.

внутр_длина

Числовая константа, задающая размер внутреннего представления нового типа в байтах. По умолчанию предполагается, что тип имеет переменную длину.

выравнивание

Требуемое выравнивание для типа данных. Допустимые значения этого параметра, если он указывается: `char`, `int2`, `int4` или `double`; по умолчанию подразумевается `int4`.

хранение

Стратегия хранения для типа данных. Допустимые значения этого параметра, если он указывается: `plain`, `external`, `extended` или `main`; по умолчанию подразумевается `plain`.

тип_образец

Имя существующего типа данных, от которого новый тип получит свойства представления. Из этого типа будут скопированы значения параметров *internallength*, *passedbyvalue*, *alignment* и *storage*, если их не переопределят явные указания, заданные дополнительно в этой команде `CREATE TYPE`.

категория

Код категории (один символ ASCII) для этого типа. По умолчанию подразумевается `'U'` (что означает пользовательский тип, «User-defined»). Коды других стандартных категорий можно найти в Таблице 49.63. Для нестандартных категорий можно выбрать другие ASCII-символы.

предпочитаемый

Если значение этого параметра равно true, создаваемый тип будет предпочитаемым в своей категории. По умолчанию подразумевается false. Будьте очень осторожны, создавая новый предпочитаемый тип в существующей категории, так как это может поменять поведение выражений неожиданным образом.

по_умолчанию

Значение по умолчанию для создаваемого типа данных. Если не указано, значением по умолчанию будет NULL.

элемент

Создаваемый тип будет массивом; этот параметр определяет тип элементов массива.

разделитель

Символ, разделяющий значения в массивах, образованных из значений создаваемого типа.

сортируемый

Если значение этого параметра равно true, в операциях с создаваемым типом может учитываться информация о правилах сортировки. По умолчанию подразумевается false.

Замечания

Так как на использование типа данных после создания не накладываются ограничения, объявление базового типа или типа-диапазона по сути даёт всем право на выполнение функций, упомянутых в определении типа. Обычно это не проблема для таких функций, какие бывают полезны в определении типов. Но прежде чем создать тип, преобразование которого во внешнюю форму и обратно будет использовать «секретную» информацию, стоит подумать дважды.

В PostgreSQL до версии 8.3 имя генерируемого типа-массива всегда образовалось из имени типа элемента и добавленного спереди символа подчёркивания (_). (Таким образом, допустимая максимальная длина имени типа была на символ меньше, чем длины других имён.) Хотя и сейчас имя типа массива чаще всего образуется таким образом, оно может быть и другим в

случае достижения максимальной длины или конфликтов с именами пользовательских типов, начинающихся с подчёркивания. Поэтому полагаться на это соглашение в коде не рекомендуется. Вместо этого, имя типа массива, связанного с данным типом, следует определять по значению pg_type.typarray.

Вообще же можно посоветовать не использовать имена типов и таблиц, начинающиеся с подчёркивания. Хотя сервер сможет сгенерировать другое имя, не конфликтующее с пользовательским, некоторая путаница всё же возможна, особенно со старыми клиентскими приложениями, которые могут полагать, что имя типа, начинающееся с подчёркивания, всегда относится к типу массива.

В PostgreSQL до версии 8.2 у CREATE TYPE *имя* отсутствовала форма для создания типа-пустышки. Поэтому для создания нового базового типа требовалось сначала создать функцию ввода. При таком подходе Postgres Pro воспринимал тип возврата функции ввода как имя нового типа данных и неявно создавал тип-пустышку, на который затем можно было ссылаться в определениях остальных функций ввода/вывода. Этот подход по-прежнему работает, но считается устаревшим и может быть запрещён в будущих версиях. Кроме того, во избежание непреднамеренного заполнения каталогов типами-пустышками, появляющимися в результате простых опечаток в определении функций, тип-пустышка будет создаваться таким образом, только если функция ввода написана на C.

В PostgreSQL до версии 7.3 было принято вовсе не создавать тип-пустышку, заменяя в определении функций ссылки на ещё не созданный тип именем псевдотипа opaque. Аргументы cstring и результаты так же должны были объявляться как opaque до версии 7.3. Для поддержки загрузки старых файлов экспорта БД, CREATE TYPE примет ссылки на функции ввода/вывода, объявленные с типом opaque, но при этом выдаст замечание и изменит в объявлении функции псевдотип на правильный.

Примеры

В этом примере создаётся составной тип, а затем он используется в определении функции:

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
    SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

В этом примере создаётся тип-перечисление, а затем он используется в определении таблицы:

```
CREATE TYPE bug_status AS ENUM ('new', 'open', 'closed');

CREATE TABLE bug (
    id serial,
    description text,
    status bug_status
);
```

В этом примере создаётся тип-диапазон:

```
CREATE TYPE float8_range AS RANGE (subtype = float8, subtype_diff = float8mi);
```

В следующем примере создаётся базовый тип данных box, а затем он используется в определении таблицы:

```
CREATE TYPE box;

CREATE FUNCTION my_box_in_function(cstring) RETURNS box AS ... ;
CREATE FUNCTION my_box_out_function(box) RETURNS cstring AS ... ;

CREATE TYPE box (
```

```
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

Если бы внутренней структурой `box` был массив из четырёх элементов `float4`, вместо этого можно было бы использовать определение:

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function,
    ELEMENT = float4
);
```

В таком случае к числам, составляющим значение этого типа, можно было бы обращаться по индексу. В остальном поведение этого типа будет таким же.

В этом примере создаётся тип большого объекта, а затем он используется в определении таблицы:

```
CREATE TYPE bigobj (
    INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE
);
CREATE TABLE big_objs (
    id integer,
    obj bigobj
);
```

Другие примеры, в том числе демонстрирующие подходящие функции ввода/вывода, можно найти в Разделе 35.11.

Совместимость

Первая форма команды `CREATE TYPE`, создающая составной тип, соответствует стандарту SQL. Другие формы являются расширениями Postgres Pro. Для оператора `CREATE TYPE` в стандарте SQL также определены другие формы, не реализованные в Postgres Pro.

Возможность создавать составной тип без атрибутов — специфическое отклонение Postgres Pro от стандарта (как и аналогичная особенность команды `CREATE TABLE`).

См. также

ALTER TYPE, CREATE DOMAIN, CREATE FUNCTION, DROP TYPE

CREATE USER

CREATE USER — создать роль в базе данных

Синтаксис

```
CREATE USER имя [ [ WITH ] параметр [ ... ] ]

Здесь параметр:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT предел_подключений
    | [ ENCRYPTED ] PASSWORD 'пароль'
    | VALID UNTIL 'дата_время'
    | IN ROLE имя_роли [, ...]
    | IN GROUP имя_роли [, ...]
    | ROLE имя_роли [, ...]
    | ADMIN имя_роли [, ...]
    | USER имя_роли [, ...]
    | SYSID uid
```

Описание

Команда `CREATE USER` теперь является просто синонимом CREATE ROLE. Единственное отличие в том, что для команды, записанной в виде `CREATE USER`, по умолчанию подразумевается `LOGIN`, а в виде `CREATE ROLE` подразумевается `NOLOGIN`.

Совместимость

Оператор `CREATE USER` является расширением Postgres Pro. В стандарте SQL определение пользователей считается зависимым от реализации.

См. также

CREATE ROLE

CREATE USER MAPPING

CREATE USER MAPPING — создать сопоставление пользователя для стороннего сервера

Синтаксис

```
CREATE USER MAPPING [IF NOT EXISTS] FOR { имя_пользователя | USER | CURRENT_USER |
 PUBLIC }
    SERVER имя_сервера
    [ OPTIONS ( параметр 'значение' [ , ... ] ) ]
```

Описание

`CREATE USER MAPPING` создаёт сопоставление пользователя на внешнем сервере. Сопоставление пользователя обычно содержит информацию о подключении, которую будет использовать обёртка сторонних данных вместе с информацией о стороннем сервере для получения доступа к внешнему ресурсу.

Владелец стороннего сервера может создать сопоставление для любых пользователей на этом сервере. Кроме того, пользователь может создать сопоставление для своего собственного имени пользователя, если он наделён правом `USAGE` на данном сервере.

Параметры

`IF NOT EXISTS`

Не считать ошибкой, если сопоставление данного пользователя для данного стороннего сервера уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее сопоставление как-то соотносится с тем, которое могло бы быть создано.

имя_пользователя

Имя существующего пользователя, для которого создаётся сопоставление на стороннем сервере. Ключевые слова `CURRENT_USER` и `USER` обозначают имя текущего пользователя. Если указывается `PUBLIC`, создаётся так называемое общее сопоставление, которое будет использоваться при отсутствии сопоставления для конкретного пользователя.

имя_сервера

Имя существующего сервера, для которого создаётся сопоставление пользователя.

`OPTIONS ( параметр 'значение' [, ... ] )`

В этом предложении задаются параметры сопоставления. Эти параметры обычно определяют фактическое имя и пароль пользователя на целевом сервере. Имена параметров должны быть уникальными. Набор допустимых имён и значений параметров определяется обёрткой сторонних данных внешнего сервера.

Примеры

Создание сопоставления для пользователя `bob` на сервере `foo`:

```
CREATE USER MAPPING FOR bob SERVER foo OPTIONS (user 'bob', password 'secret');
```

Совместимость

`CREATE USER MAPPING` соответствует стандарту ISO/IEC 9075-9 (SQL/MED).

См. также

ALTER USER MAPPING, DROP USER MAPPING, CREATE FOREIGN DATA WRAPPER, CREATE SERVER

CREATE VIEW

CREATE VIEW — создать представление

Синтаксис

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW имя [ ( имя_столбца
 [, ...] ) ]
    [ WITH ( имя_параметра_представления [= значение_параметра_представления]
 [, ... ] ) ]
    AS запрос
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Описание

`CREATE VIEW` создаёт представление запроса. Создаваемое представление лишено физической материализации, поэтому указанный запрос будет выполняться при каждом обращении к представлению.

Команда `CREATE OR REPLACE VIEW` действует подобным образом, но если представление с этим именем уже существует, оно заменяется. Новый запрос должен выдавать те же столбцы, что выдавал запрос, ранее определённый для этого представления (то есть, столбцы с такими же именами должны иметь те же типы данных и следовать в том же порядке), но может добавить несколько новых столбцов в конце списка. Вычисления, в результате которых формируются столбцы представления, могут быть совершенно другими.

Если задано имя схемы (например, `CREATE VIEW myschema.myview ...`), представление создаётся в указанной схеме, в противном случае — в текущей. Временные представления существуют в специальной схеме, так что при создании таких представлений имя схемы задать нельзя. Имя представления должно отличаться от имён других представлений, таблиц, последовательностей, индексов или сторонних таблиц в этой схеме.

Параметры

`TEMPORARY` или `TEMP`

С таким указанием представление создаётся как временное. Временные представления автоматически удаляются в конце сеанса. Существующее постоянное представление с тем же именем не будет видно в текущем сеансе, пока существует временное, однако к нему можно обратиться, дополнив имя указанием схемы.

Если в определении представления задействованы временные таблицы, представление так же создаётся как временное (вне зависимости от присутствия явного указания `TEMPORARY`).

`RECURSIVE`

Создаёт рекурсивное представление. Синтаксис

```
CREATE RECURSIVE VIEW [ схема . ] имя (имена_столбцов) AS SELECT ...;
```

равнозначен

```
CREATE VIEW [ схема . ] имя AS WITH RECURSIVE имя (имена_столбцов) AS (SELECT ...)
 SELECT имена_столбцов FROM имя;
```

Для рекурсивного представления обязательно должен задаваться список с именами столбцов.

имя

Имя создаваемого представления (возможно, дополненное схемой).

имя_столбца

Необязательный список имён, назначаемых столбцам представления. Если отсутствует, имена столбцов формируются из результатов запроса.

`WITH ( имя_параметра_представления [= значение_параметра_представления] [, ... ] )`

В этом предложении могут задаваться следующие необязательные параметры представления:

`check_option` (`string`)

Этот параметр может принимать значение `local` (локально) или `cascaded` (каскадно) и равнозначен указанию `WITH [ CASCADED | LOCAL ] CHECK OPTION` (см. ниже). Изменить этот параметр у существующего представления с помощью ALTER VIEW нельзя.

`security_barrier` (`boolean`)

Этот параметр следует использовать, если представление должно обеспечивать защиту на уровне строк. За дополнительными подробностями обратитесь к Разделу 38.5.

запрос

Команда SELECT или VALUES, которая выдаёт столбцы и строки представления.

`WITH [ CASCADED | LOCAL ] CHECK OPTION`

Это указание управляет поведением автоматически изменяемых представлений. Если оно присутствует, при выполнении операций `INSERT` и `UPDATE` с этим представлением будет проверяться, удовлетворяют ли новые строки условию, определяющему представление (то есть, проверяется, будут ли новые строки видны через это представление). Если они не удовлетворяют условию, операция не будет выполнена. Если указание `CHECK OPTION` отсутствует, команды `INSERT` и `UPDATE` смогут создавать в этом представлении строки, которые не будут видны в нём. Поддерживаются следующие варианты проверки:

`LOCAL`

Новые строки проверяются только по условиям, определённым непосредственно в самом представлении. Любые условия, определённые в нижележащих базовых представлениях, не проверяются (если только в них нет указания `CHECK OPTION`).

`CASCADED`

Новые строки проверяются по условиям данного представления и всех нижележащих базовых. Если указано `CHECK OPTION`, а `LOCAL` и `CASCADED` опущено, подразумевается указание `CASCADED`.

Указание `CHECK OPTION` нельзя использовать с рекурсивными представлениями.

Заметьте, что `CHECK OPTION` поддерживается только для автоматически изменяемых представлений, не имеющих триггеров `INSTEAD OF` и правил `INSTEAD`. Если автоматически изменяемое представление определено поверх базового представления с триггерами `INSTEAD OF`, то для проверки ограничений автоматически изменяемого представления можно применить указание `LOCAL CHECK OPTION`, хотя условия базового представления с триггерами `INSTEAD OF` при этом проверяться не будут (каскадная проверка не будет спускаться к представлению, модифицируемому триггером, и любые параметры проверки, определённые для такого представления, будут просто игнорироваться). Если для представления или любого из его базовых отношений определено правило `INSTEAD`, приводящее к перезаписи команды `INSERT` или `UPDATE`, в перезаписанном запросе все параметры проверки будут игнорироваться, в том числе проверки автоматически изменяемых представлений, определённых поверх отношений с правилом `INSTEAD`.

Замечания

Для удаления представлений применяется оператор DROP VIEW.

Позаботьтесь о том, чтобы столбцы представления получили желаемые имена и типы. Например, такая команда:

```
CREATE VIEW vista AS SELECT 'Hello World';
```

плоха тем, что именем столбца по умолчанию будет `?column?`, а типом данных — `text`; и это может быть не совсем то, чего вы хотите. Лучше записывать строковую константу в результате представления примерно так:

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

Доступ к таблицам, задействованным в представлении, определяется правами владельца представления. В некоторых случаях это позволяет организовать безопасный, но ограниченный доступ к нижележащим таблицам. Однако учтите, что не все представления могут быть защищёнными; за подробностями обратитесь к Разделу 38.5. Функции, вызываемые в представлении, выполняются так, как будто они вызываются непосредственно из запроса, обращающегося к представлению. Поэтому пользователь представления должен иметь все права, необходимые для вызова всех функций, задействованных в представлении.

При выполнении `CREATE OR REPLACE VIEW` для существующего представления меняется только правило SELECT, определяющее представление. Другие свойства представления, включая владельца, права и правила, кроме SELECT, остаются неизменными. Чтобы изменить определение представления, необходимо быть его владельцем (или членом роли-владельца).

Изменяемые представления

Простые представления становятся изменяемыми автоматически: система позволит выполнять команды `INSERT`, `UPDATE` и `DELETE` с таким представлением так же, как и с обычной таблицей. Представление будет автоматически изменяемым, если оно удовлетворяют одновременно всем следующим условиям:

- Список `FROM` в запросе, определяющем представлении, должен содержать ровно один элемент, и это должна быть таблица или другое изменяемое представление.
- Определение представления не должно содержать предложения `WITH`, `DISTINCT`, `GROUP BY`, `HAVING`, `LIMIT` и `OFFSET` на верхнем уровне запроса.
- Определение представления не должно содержать операции с множествами (`UNION`, `INTERSECT` и `EXCEPT`) на верхнем уровне запроса.
- Список выборки в запросе не должен содержать агрегатные и оконные функции, а также функции, возвращающие множества.

Автоматически обновляемое представление может содержать как изменяемые, так и не изменяемые столбцы. Столбец будет изменяемым, если это простая ссылка на изменяемый столбец нижележащего базового отношения; в противном случае этот столбец будет доступен только для чтения, и если команда `INSERT` или `UPDATE` попытается записать значение в него, возникнет ошибка.

Если представление автоматически изменяемое, система будет преобразовывать обращающиеся к нему операторы `INSERT`, `UPDATE` и `DELETE` в соответствующие операторы, обращающиеся к нижележащему базовому отношению. При этом в полной мере поддерживаются операторы `INSERT` с предложением `ON CONFLICT UPDATE`.

Если автоматически изменяемое представление содержит условие `WHERE`, это условие ограничивает набор строк, которые могут быть изменены командой `UPDATE` и удалены командой `DELETE` в этом представлении. Однако `UPDATE` может изменить строку так, что она больше не будет соответствовать условию `WHERE` и, как следствие, больше не будет видна через представление. Команда `INSERT` подобным образом может вставить в базовое отношение строки, которые не удовлетворят условию `WHERE` и поэтому не будут видны через представление (`ON CONFLICT UPDATE` может подобным образом воздействовать на существующую строку, не видимую через представление). Чтобы запретить командам `INSERT` и `UPDATE` создавать такие строки, которые не видны через представление, можно воспользоваться указанием `CHECK OPTION`.

Если автоматически изменяемое представление имеет свойство `security_barrier` (барьер безопасности), то все условия `WHERE` этого представления (и все условия с герметичными операторами (`LEAKPROOF`)) будут всегда вычисляться перед условиями, добавленными пользователем представления. За подробностями обратитесь к Разделу 38.5. Заметьте, что по этой причине строки, которые в конце концов не были выданы (потому что не прошли проверку в пользовательском условии `WHERE`), могут всё же остаться заблокированными. Чтобы определить, какие условия применяются на уровне отношения (и, как следствие, избавляют часть строк от блокировки), можно воспользоваться командой `EXPLAIN`.

Более сложные представления, не удовлетворяющие этим условиям, по умолчанию доступны только для чтения: система не позволит выполнить операции добавления, изменения или удаления строк в таком представлении. Создать эффект изменяемого представления для них можно, определив триггеры `INSTEAD OF`, которые будут преобразовывать запросы на изменение данных в соответствующие действия с другими таблицами. За дополнительными сведениями обратитесь к CREATE TRIGGER. Так же есть возможность создавать правила (см. CREATE RULE), но на практике триггеры проще для понимания и применения.

Учтите, что пользователь, выполняющий операции добавления, изменения или удаления данных в представлении, должен иметь соответствующие права для этого представления. Кроме того, владелец представления должен иметь сопутствующие права в нижележащих базовых отношениях, хотя пользователь, собственно выполняющий эти операции, может этих прав не иметь (см. Раздел 38.5).

Примеры

Создание представления, содержащего все комедийные фильмы:

```
CREATE VIEW comedies AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy';
```

Эта команда создаст представление со столбцами, которые содержались в таблице `film` в момент выполнения команды. Хотя при создании представления было указано `*`, столбцы, добавляемые в таблицу позже, частью представления не будут.

Создание представления с указанием `LOCAL CHECK OPTION`:

```
CREATE VIEW universal_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'U'
    WITH LOCAL CHECK OPTION;
```

Эта команда создаст представление на базе представления `comedies`, выдающее только комедии (`kind = 'Comedy'`) универсальной возрастной категории `classification = 'U'`. Любая попытка выполнить в представлении `INSERT` или `UPDATE` со строкой, не удовлетворяющей условию `classification = 'U'`, будет отвергнута, но ограничение по полю `kind` (тип фильма) проверяться не будет.

Создание представления с указанием `CASCADED CHECK OPTION`:

```
CREATE VIEW pg_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'PG'
    WITH CASCADED CHECK OPTION;
```

Это представление будет проверять, удовлетворяют ли новые строки обоим условиям: по столбцу `kind` и по столбцу `classification`.

Создание представления с изменяемыми и неизменяемыми столбцами:

```
CREATE VIEW comedies AS
    SELECT f.*,
           country_code_to_name(f.country_code) AS country,
           (SELECT avg(r.rating)
            FROM user_ratings r
            WHERE r.film_id = f.id) AS avg_rating
    FROM films f
    WHERE f.kind = 'Comedy';
```

Это представление будет поддерживать операции `INSERT`, `UPDATE` и `DELETE`. Изменяемыми будут все столбцы из таблицы `films`, тогда как вычисляемые столбцы `country` и `avg_rating` будут доступны только для чтения.

Создание рекурсивного представления, содержащего числа от 1 до 100:

```
CREATE RECURSIVE VIEW public.nums_1_100 (n) AS
    VALUES (1)
UNION ALL
    SELECT n+1 FROM nums_1_100 WHERE n < 100;
```

Заметьте, что несмотря на то, что имя рекурсивного представления дополнено схемой в этой команде `CREATE`, внутренняя ссылка представления на себя же схемой не дополняется. Это связано с тем, что имя неявно создаваемого CTE не может дополняться схемой.

Совместимость

Команда `CREATE OR REPLACE VIEW` — языковое расширение Postgres Pro. Так же расширением является предложение `WITH ( ... )` и концепция временного представления.

См. также

ALTER VIEW, DROP VIEW, CREATE MATERIALIZED VIEW

DEALLOCATE

DEALLOCATE — освободить подготовленный оператор

Синтаксис

```
DEALLOCATE [ PREPARE ] { имя | ALL }
```

Описание

`DEALLOCATE` применяется для освобождения ранее подготовленного оператора SQL. Если не освободить подготовленный оператор явно, он будет освобождён при завершении сеанса.

За дополнительными сведениями о подготовленных операторах обратитесь к PREPARE.

Параметры

`PREPARE`

Это ключевое слово игнорируется.

имя

Имя подготовленного оператора, подлежащего освобождению.

`ALL`

Освобождает все подготовленные операторы.

Совместимость

В стандарте SQL есть оператор `DEALLOCATE`, но он предназначен только для применения во встраиваемом SQL.

См. также

EXECUTE, PREPARE

DECLARE

DECLARE — определить курсор

Синтаксис

```
DECLARE имя [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
    CURSOR [ { WITH | WITHOUT } HOLD ] FOR запрос
```

Описание

Оператор `DECLARE` позволяет пользователю создавать курсоры, с помощью которых можно выбирать по очереди некоторое количество строк из результата большого запроса. Когда курсор создан, через него можно получать строки, применяя команду FETCH.

> **Примечание**
>
> На этой странице описывается применение курсоров на уровне команд SQL. Если вы попытаетесь использовать курсоры внутри функции PL/pgSQL, правила будут другими — см. Раздел 40.7.

Параметры

имя

Имя создаваемого курсора.

`BINARY`

Курсор с таким свойством возвращает данные в двоичном, а не текстовом формате.

`INSENSITIVE`

Указывает, что данные, считываемые из курсора, не должны зависеть от изменений, которые могут происходить в нижележащих таблицах после создания курсора. В Postgres Pro это поведение подразумевается по умолчанию, так что это ключевое слово ни на что не влияет и принимается только для совместимости со стандартом SQL.

`SCROLL`
`NO SCROLL`

Указание `SCROLL` определяет, что курсор может прокручивать набор данных и получать строки непоследовательно (например, в обратном порядке). В зависимости от сложности плана запроса указание `SCROLL` может отрицательно отразиться на скорости выполнения запроса. Указание `NO SCROLL`, напротив, определяет, что через курсор нельзя будет получать строки в произвольном порядке. По умолчанию прокрутка в некоторых случаях разрешается; но это не равнозначно эффекту указания `SCROLL`. За подробностями обратитесь к Разделу «Замечания».

`WITH HOLD`
`WITHOUT HOLD`

Указание `WITH HOLD` определяет, что курсор можно продолжать использовать после успешной фиксации создавшей его транзакции. `WITHOUT HOLD` определяет, что курсор нельзя будет использовать за рамками транзакции, создавшей его. Если не указано ни `WITHOUT HOLD`, ни `WITH HOLD`, по умолчанию подразумевается `WITHOUT HOLD`.

запрос

Команда SELECT или VALUES, выдающая строки, которые будут получены через курсор.

Ключевые слова BINARY, INSENSITIVE и SCROLL могут указываться в любом порядке.

Замечания

Обычный курсор выдаёт данные в текстовом виде, в каком их выдаёт SELECT. Однако с указанием BINARY курсор может выдавать их и в двоичном формате. Это упрощает операции преобразования данных для сервера и клиента, за счёт дополнительных усилий, требующихся от программиста для работы с платформозависимыми двоичными форматами. Например, если запрос получает значение 1 из целочисленного столбца, обычный курсор выдаст строку, содержащую 1, тогда как через двоичный курсор будет получено четырёхбайтовое поле, содержащее внутреннее представление значения (с сетевым порядком байтов).

Двоичные курсоры должны применяться с осмотрительностью. Многие приложения, в том числе psql, не приспособлены к работе с двоичными курсорами и ожидают, что данные будут поступать в текстовом формате.

Примечание

Когда клиентское приложение выполняет команду FETCH, используя протокол «расширенных запросов», в сообщении Bind этого протокола указывается, в каком формате, текстовом или двоичном, должны быть получены данные. Это указание переопределяет свойство курсора, заданное в его объявлении. Таким образом, концепция курсора, объявляемого двоичным, становится устаревшей при использовании протокола расширенных запросов — любой курсор может быть прочитан как текстовый или двоичный.

Если в команде объявления курсора не указано WITH HOLD, созданный ей курсор может использоваться только в текущей транзакции. Таким образом, оператор DECLARE без WITH HOLD бесполезен вне блока транзакции: курсор будет существовать только до завершения этого оператора. Поэтому Postgres Pro сообщает об ошибке, если такая команда выполняется вне блока транзакции. Чтобы определить блок транзакции, примените команды BEGIN и COMMIT (или ROLLBACK).

Если в объявлении курсора указано WITH HOLD и транзакция, создавшая курсор, успешно фиксируется, к этому курсору могут продолжать обращаться последующие транзакции в этом сеансе. (Но если создавшая курсор транзакция прерывается, курсор уничтожается.) Курсор со свойством WITH HOLD (удерживаемый) может быть закрыт явно, командой CLOSE, либо неявно, по завершении сеанса. В текущей реализации строки, представляемые удерживаемым курсором, копируются во временный файл или в область памяти, так что они остаются доступными для следующих транзакций.

Объявить курсор со свойством WITH HOLD можно, только если запрос не содержит указаний FOR UPDATE и FOR SHARE.

Указание SCROLL добавляется при определении курсора, который будет выбирать данные в обратном порядке. Это поведение требуется стандартом SQL. Однако для совместимости с предыдущими версиями, Postgres Pro допускает выборку в обратном направлении и без указания SCROLL, если план запроса курсора достаточно прост, чтобы реализовать прокрутку назад без дополнительных операций. Тем не менее разработчикам приложений не следует рассчитывать на то, что курсор, созданный без указания SCROLL, можно будет прокручивать назад. С указанием NO SCROLL прокрутка назад запрещается в любом случае.

Выборка в обратном направлении также запрещается, если запрос содержит указания FOR UPDATE и FOR SHARE; в этом случае указание SCROLL не принимается.

Внимание

Прокручиваемые и удерживаемые (WITH HOLD) курсоры могут выдавать неожиданные результаты, если они вызывают изменчивые функции (см. Раздел 35.6). Когда повторно

выбирается ранее прочитанная строка, функции могут вызываться снова и выдавать результаты, отличные от полученных в первый раз. Один из способов обойти эту проблему — объявить курсор с указанием WITH HOLD и зафиксировать транзакцию, прежде чем читать из него какие-либо строки. В этом случае весь набор данных курсора будет материализован во временном хранилище, так что изменчивые функции будут выполнены для каждой строки лишь единожды.

Если запрос в определении курсора включает указания FOR UPDATE или FOR SHARE, возвращаемые курсором строки блокируются в момент первой выборки, так же, как это происходит при выполнении SELECT с этими указаниями. Кроме того, при чтении строк будут возвращаться их наиболее актуальные версии; таким образом, с этими указаниями курсор будет вести себя как «чувствительный курсор», определённый в стандарте SQL. (Указать INSENSITIVE для курсора с запросом FOR UPDATE или FOR SHARE нельзя.)

Внимание

Обычно рекомендуется использовать FOR UPDATE, если курсор предназначается для применения в командах UPDATE ... WHERE CURRENT OF и DELETE ... WHERE CURRENT OF. Указание FOR UPDATE предотвращает изменение строк другими сеансами после того, как они были считаны, и до того, как выполнится команда. Без FOR UPDATE последующая команда с WHERE CURRENT OF не сработает, если строка будет изменена после создания курсора.

Ещё одна причина использовать указание FOR UPDATE в том, что без него последующие команды с WHERE CURRENT OF могут выдать ошибку, если запрос курсора не удовлетворяет оговоренному в стандарте SQL критерию «простой изменяемости» (в частности, курсор должен ссылаться только на одну таблицу и не должен использовать группировку и сортировку (ORDER BY)). Курсоры, не удовлетворяющие этому критерию, могут работать либо не работать, в зависимости от конкретного выбранного плана; так что в худшем случае приложение может работать в тестовой, но сломается в производственной среде. С указанием FOR UPDATE курсор гарантированно будет изменяемым.

Не использовать же FOR UPDATE для команд с WHERE CURRENT OF в основном имеет смысл, только если требуется получить прокручиваемый курсор или курсор, не отражающий последующие изменения (то есть, продолжающий показывать прежние данные). Если это действительно необходимо, обязательно учтите при реализации приведённые выше замечания.

В стандарте SQL механизм курсоров предусмотрен только для встраиваемого SQL. Сервер Postgres Pro не реализует для курсоров оператор OPEN; курсор считается открытым при объявлении. Однако ECPG, встраиваемый препроцессор SQL для Postgres Pro, следует соглашениям стандарта, в том числе поддерживая для курсоров операторы DECLARE и OPEN.

Получить список всех доступных курсоров можно, обратившись к системному представлению pg_cursors.

Примеры

Объявление курсора:

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

Другие примеры использования курсора можно найти в FETCH.

Совместимость

В стандарте SQL говорится, что чувствительность курсоров к параллельному обновлению нижележащих данных по умолчанию определяется реализацией. В Postgres Pro курсоры по

умолчанию нечувствительные, а чувствительными их можно сделать с помощью указания `FOR UPDATE`. Другие СУБД могут работать иначе.

Стандарт SQL допускает курсоры только во встраиваемом SQL и в модулях. Postgres Pro позволяет использовать курсоры интерактивно.

Двоичные курсоры являются расширением Postgres Pro.

См. также

CLOSE, FETCH, MOVE

DELETE

DELETE — удалить записи таблицы

Синтаксис

```
[ WITH [ RECURSIVE ] запрос_WITH [, ...] ]
DELETE FROM [ ONLY ] имя_таблицы [ * ] [ [ AS ] псевдоним ]
    [ USING элемент_FROM [, ...] ]
    [ WHERE условие | WHERE CURRENT OF имя_курсора ]
    [ RETURNING * | выражение_результата [ [ AS ] имя_результата ] [, ...] ]
```

Описание

Команда DELETE удаляет из указанной таблицы строки, удовлетворяющие условию WHERE. Если предложение WHERE отсутствует, она удаляет из таблицы все строки, в результате будет получена рабочая, но пустая таблица.

> **Подсказка**
>
> TRUNCATE реализует более быстрый механизм удаления всех строк из таблицы.

Удалить строки в таблице, используя информацию из других таблиц в базе данных, можно двумя способами: применяя вложенные запросы или указав дополнительные таблицы в предложении USING. Выбор предпочитаемого варианта зависит от конкретных обстоятельств.

Предложение RETURNING указывает, что команда DELETE должна вычислить и возвратить значения для каждой фактически удалённой строки. Вычислить в нём можно любое выражение со столбцами целевой таблицы и/или столбцами других таблиц, упомянутых в USING. Список RETURNING имеет тот же синтаксис, что и список результатов SELECT.

Чтобы удалять данные из таблицы, необходимо иметь право DELETE для неё, а также право SELECT для всех таблиц, перечисленных в предложении USING, и таблиц, данные которых считываются в *условии*.

Параметры

запрос_WITH

Предложение WITH позволяет задать один или несколько подзапросов, на которые затем можно ссылаться по имени в запросе DELETE. Подробнее об этом см. Раздел 7.8 и SELECT.

имя_таблицы

Имя (возможно, дополненное схемой) таблицы, из которой будут удалены строки. Если перед именем таблицы добавлено ONLY, соответствующие строки удаляются только из указанной таблицы. Без ONLY строки будут также удалены из всех таблиц, унаследованных от указанной. При желании, после имени таблицы можно указать *, чтобы явно обозначить, что операция затрагивает все дочерние таблицы.

псевдоним

Альтернативное имя целевой таблицы. Когда указывается это имя, оно полностью скрывает фактическое имя таблицы. Например, в запросе DELETE FROM foo AS f дополнительные компоненты оператора DELETE должны обращаться к целевой таблице по имени f, а не foo.

элемент_FROM

Табличное выражение, позволяющее добавить в условие WHERE столбцы из других таблиц. В этом выражении используется тот же синтаксис, что и в предложении «Предложение FROM»

оператора SELECT; например, в нём можно определить псевдоним для таблицы. Повторять в нём имя целевой таблицы нужно, только если требуется определить замкнутое соединение (в этом случае для данного имени должен определяться псевдоним).

условие

Выражение, возвращающее значение типа boolean. Удалены будут только те строки, для которых это выражение возвращает true.

имя_курсора

Имя курсора, который будет использоваться в условии WHERE CURRENT OF. С таким условием будет удалена строка, выбранная из этого курсора последней. Курсор должен образовываться запросом, не применяющим группировку, к целевой таблице команды DELETE. Заметьте, что WHERE CURRENT OF нельзя задать вместе с логическим условием. За дополнительными сведениями об использовании курсоров с WHERE CURRENT OF обратитесь к DECLARE.

выражение_результата

Выражение, которое будет вычисляться и возвращаться командой DELETE после удаления каждой строки. В этом выражении можно использовать имена любых столбцов таблицы *имя_таблицы* или таблиц, перечисленных в списке USING. Чтобы получить все столбцы, достаточно написать *.

имя_результата

Имя, назначаемое возвращаемому столбцу.

Выводимая информация

В случае успешного завершения, DELETE возвращает метку команды в виде

DELETE *число*

Здесь *число* — количество удалённых строк. Заметьте, что это число может быть меньше числа строк, соответствующих *условию*, если удаления были подавлены триггером BEFORE DELETE. Если *число* равно 0, это означает, что запрос не удалил ни одной строки (это не считается ошибкой).

Если команда DELETE содержит предложение RETURNING, её результат будет похож на результат оператора SELECT (с теми же столбцами и значениями, что содержатся в списке RETURNING), полученный для строк, удалённых этой командой.

Замечания

Postgres Pro позволяет ссылаться на столбцы других таблиц в условии WHERE, когда эти таблицы перечисляются в предложении USING. Например, удалить все фильмы определённого продюсера можно так:

```
DELETE FROM films USING producers
  WHERE producer_id = producers.id AND producers.name = 'foo';
```

По сути в этом запросе выполняется соединение таблиц films и producers, и все успешно включённые в соединение строки в films помечаются для удаления. Этот синтаксис не соответствует стандарту. Следуя стандарту, эту задачу можно решить так:

```
DELETE FROM films
  WHERE producer_id IN (SELECT id FROM producers WHERE name = 'foo');
```

В ряде случаев запрос в стиле соединения легче написать и он может работать быстрее, чем в стиле вложенного запроса.

Примеры

Удаление всех фильмов, кроме мюзиклов:

```
DELETE FROM films WHERE kind <> 'Musical';
```

Очистка таблицы `films`:

```
DELETE FROM films;
```

Удаление завершённых задач с получением всех данных удалённых строк:

```
DELETE FROM tasks WHERE status = 'DONE' RETURNING *;
```

Удаление из `tasks` строки, на которой в текущий момент располагается курсор `c_tasks`:

```
DELETE FROM tasks WHERE CURRENT OF c_tasks;
```

Совместимость

Эта команда соответствует стандарту SQL, но предложения `USING` и `RETURNING` являются расширениями Postgres Pro, как и возможность использовать `WITH` с `DELETE`.

См. также

TRUNCATE

DISCARD

DISCARD — очистить состояние сеанса

Синтаксис

```
DISCARD { ALL | PLANS | SEQUENCES | TEMPORARY | TEMP }
```

Описание

`DISCARD` высвобождает внутренние ресурсы, связанные с сеансом использования базы данных. Эта команда полезна для частичного или полного сброса состояния сеанса. Для освобождения различных типов ресурсов она поддерживает несколько разных подкоманд; вариант `DISCARD ALL` включает в себя все остальные и также сбрасывает дополнительное состояние.

Параметры

`PLANS`

Высвобождает все кешированные планы запросов, вынуждая сервер провести планирование заново при следующем использовании связанного подготовленного оператора.

`SEQUENCES`

Сбрасывает кешированное состояние, связанное с последовательностями, включая внутреннюю информацию `currval()`/`lastval()` и любые предварительно выделенные значения последовательностей, которые ещё не выдала функция `nextval()`. (Кеширование значений последовательности описано в CREATE SEQUENCE.)

`TEMPORARY` или `TEMP`

Удаляет все временные таблицы, созданные в текущем сеансе.

`ALL`

Высвобождает все временные ресурсы, связанные с текущим сеансом, и сбрасывает сеанс к начальному состоянию. В настоящее время действует так же, как и следующая последовательность операторов:

```
SET SESSION AUTHORIZATION DEFAULT;
RESET ALL;
DEALLOCATE ALL;
CLOSE ALL;
UNLISTEN *;
SELECT pg_advisory_unlock_all();
DISCARD PLANS;
DISCARD SEQUENCES;
DISCARD TEMP;
```

Замечания

`DISCARD ALL` нельзя выполнять внутри блока транзакции.

Совместимость

`DISCARD` является расширением Postgres Pro.

DO

DO — выполнить анонимный блок кода

Синтаксис

```
DO [ LANGUAGE имя_языка ] код
```

Описание

`DO` выполняет анонимный блок кода или, другими словами, разовую анонимную функцию на процедурном языке.

Блок кода воспринимается, как если бы это было тело функции, которая не имеет параметров и возвращает `void`. Этот код разбирается и выполняется один раз.

Необязательное предложение `LANGUAGE` можно записать до или после блока кода.

Параметры

код

Выполняемый код на процедурном языке. Он должен задаваться в виде текстовой строки (её рекомендуется заключать в доллары), как и код в `CREATE FUNCTION`.

имя_языка

Имя процедурного языка, на котором написан код. По умолчанию подразумевается `plpgsql`.

Замечания

Применяемый процедурный язык должен быть уже зарегистрирован в текущей базе с помощью команды `CREATE LANGUAGE`. По умолчанию зарегистрирован только `plpgsql`, но не другие языки.

Пользователь должен иметь право `USAGE` для данного процедурного языка, либо быть суперпользователем, если этот язык недоверенный. Такие же требования действуют и при создании функции на этом языке.

Примеры

Следующий код даст все права для всех представлений в схеме `public` роли `webuser`:

```
DO $$DECLARE r record;
BEGIN
    FOR r IN SELECT table_schema, table_name FROM information_schema.tables
             WHERE table_type = 'VIEW' AND table_schema = 'public'
    LOOP
        EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' ||
 quote_ident(r.table_name) || ' TO webuser';
    END LOOP;
END$$;
```

Совместимость

Оператор `DO` отсутствует в стандарте SQL.

См. также

CREATE LANGUAGE

DROP ACCESS METHOD

DROP ACCESS METHOD — удалить метод доступа

Синтаксис

```
DROP ACCESS METHOD [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP ACCESS METHOD` удаляет существующий метод доступа. Удалять методы доступа разрешено только суперпользователям.

Параметры

`IF EXISTS`

Не считать ошибкой, если метод доступа не существует. В этом случае будет выдано замечание.

имя

Имя существующего метода доступа.

`CASCADE`

Автоматически удалять объекты, зависящие от данного метода доступа (например, классы и семейства операторов, а также индексы), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении метода доступа, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление метода доступа `heptree`:

```
DROP ACCESS METHOD heptree;
```

Совместимость

`DROP ACCESS METHOD` является расширением Postgres Pro.

См. также

CREATE ACCESS METHOD

DROP AGGREGATE

DROP AGGREGATE — удалить агрегатную функцию

Синтаксис

```
DROP AGGREGATE [ IF EXISTS ] имя ( сигнатура_агр_функции ) [, ...] [ CASCADE |
 RESTRICT ]

Здесь сигнатура_агр_функции:

* |
[ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] |
[ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] ] ORDER BY
 [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ]
```

Описание

`DROP AGGREGATE` удаляет существующую агрегатную функцию. Пользователь, выполняющий эту команду, должен быть владельцем агрегатной функции.

Параметры

`IF EXISTS`

Не считать ошибкой, если агрегатная функция не существует. В этом случае будет выдано замечание.

имя

Имя существующей агрегатной функции (возможно, дополненное схемой).

режим_аргумента

Режим аргумента: `IN` или `VARIADIC`. По умолчанию подразумевается `IN`.

имя_аргумента

Имя аргумента. Заметьте, что на самом деле `DROP AGGREGATE` не обращает внимание на имена аргументов, так как для однозначной идентификации агрегатной функции достаточно только типов аргументов.

тип_аргумента

Тип входных данных, с которыми работает агрегатная функция. Чтобы сослаться на агрегатную функцию без аргументов, укажите вместо списка аргументов `*`, а чтобы сослаться на сортирующую агрегатную функцию, добавьте `ORDER BY` между указаниями непосредственных и агрегируемых аргументов.

`CASCADE`

Автоматически удалять объекты, зависящие от данной агрегатной функции (например, использующие её представления), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении агрегатной функции, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

Замечания

Альтернативные варианты указания сортирующих агрегатов описаны в ALTER AGGREGATE.

Примеры

Удаление агрегатной функции `myavg` для типа `integer`:

```
DROP AGGREGATE myavg(integer);
```

Удаление гипотезирующей агрегатной функции `myrank`, которая принимает произвольный список сортируемых столбцов и соответствующий список непосредственных аргументов:

```
DROP AGGREGATE myrank(VARIADIC "any" ORDER BY VARIADIC "any");
```

Удаление нескольких агрегатных функций одной командой:

```
DROP AGGREGATE myavg(integer), myavg(bigint);
```

Совместимость

Оператор `DROP AGGREGATE` отсутствует в стандарте SQL.

См. также

ALTER AGGREGATE, CREATE AGGREGATE

DROP CAST

DROP CAST — удалить приведение типа

Синтаксис

```
DROP CAST [ IF EXISTS ] (исходный_тип AS целевой_тип) [ CASCADE | RESTRICT ]
```

Описание

`DROP CAST` удаляет ранее определённое приведение типа.

Чтобы удалить приведение, необходимо быть владельцем исходного или целевого типа данных. Такие же требования действуют и при создании приведения.

Параметры

`IF EXISTS`

Не считать ошибкой, если приведение не существует. В этом случае будет выдано замечание.

`исходный_тип`

Имя исходного типа данных для приведения.

`целевой_тип`

Имя целевого типа данных для приведения.

`CASCADE`
`RESTRICT`

Эти ключевые слова игнорируются, так как от приведений не зависят никакие объекты.

Примеры

Удаление приведения типа `text` к типу `int`:

```
DROP CAST (text AS int);
```

Совместимость

Команда `DROP CAST` соответствует стандарту SQL.

См. также

CREATE CAST

DROP COLLATION

DROP COLLATION — удалить правило сортировки

Синтаксис

```
DROP COLLATION [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP COLLATION` удаляет ранее определённое правило сортировки. Чтобы удалить правило сортировки, необходимо быть его владельцем.

Параметры

`IF EXISTS`

Не считать ошибкой, если правило сортировки не существует. В этом случае будет выдано замечание.

имя

Имя правила сортировки, возможно, дополненное схемой.

`CASCADE`

Автоматически удалять объекты, зависящие от данного правила сортировки, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении правила сортировки, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление правила сортировки с именем `german`:

```
DROP COLLATION german;
```

Совместимость

Команда `DROP COLLATION` соответствует стандарту SQL, за исключением параметра `IF EXISTS`, являющегося расширением Postgres Pro.

См. также

ALTER COLLATION, CREATE COLLATION

DROP CONVERSION

DROP CONVERSION — удалить преобразование

Синтаксис

```
DROP CONVERSION [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP CONVERSION` удаляет ранее определённое преобразование. Чтобы удалить преобразование, необходимые быть его владельцем.

Параметры

`IF EXISTS`

Не считать ошибкой, если преобразование не существует. В этом случае будет выдано замечание.

имя

Имя преобразования, возможно, дополненное схемой.

`CASCADE`
`RESTRICT`

Эти ключевые слова игнорируются, так как от преобразований не зависят никакие объекты.

Примеры

Удаление преобразования с именем `myname`:

```
DROP CONVERSION myname;
```

Совместимость

Оператор `DROP CONVERSION` отсутствует в стандарте SQL, хотя в нём есть оператор `DROP TRANSLATION`, сопутствующий оператору `CREATE TRANSLATION`, который, в свою очередь, подобен `CREATE CONVERSION` в Postgres Pro.

См. также

ALTER CONVERSION, CREATE CONVERSION

DROP DATABASE

DROP DATABASE — удалить базу данных

Синтаксис

```
DROP DATABASE [ IF EXISTS ] имя
```

Описание

Оператор `DROP DATABASE` удаляет базу данных. Он удаляет из каталогов записи, относящиеся к базе данных, а также удаляет каталог, содержащий данные. Выполнить эту команду может только владелец базы данных. Кроме того, удалить базу нельзя, пока к ней подключены вы или кто-то ещё. (Чтобы выполнить эту команду, подключитесь к `postgres` или любой другой базе данных.)

Действие команды `DROP DATABASE` нельзя отменить. Используйте её с осторожностью!

Параметры

`IF EXISTS`

Не считать ошибкой, если база данных не существует. В этом случае будет выдано замечание.

имя

Имя базы данных, подлежащей удалению.

Замечания

`DROP DATABASE` нельзя выполнять внутри блока транзакции.

Эту команду нельзя выполнить, если установлено подключение к удаляемой базе данных. Поэтому может быть удобнее вместо неё использовать программу dropdb, которая сама вызывает эту команду внутри.

Совместимость

Оператор `DROP DATABASE` отсутствует в стандарте SQL.

См. также

CREATE DATABASE

DROP DOMAIN

DROP DOMAIN — удалить домен

Синтаксис

```
DROP DOMAIN [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP DOMAIN` удаляет домен. Удалить домен может только его владелец.

Параметры

`IF EXISTS`

Не считать ошибкой, если домен не существует. В этом случае будет выдано замечание.

имя

Имя существующего домена (возможно, дополненное схемой).

`CASCADE`

Автоматически удалять объекты, зависящие от данного домена (например, столбцы таблиц), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении домена, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление домена `box`:

```
DROP DOMAIN box;
```

Совместимость

Эта команда соответствует стандарту SQL, за исключением параметра `IF EXISTS`, являющегося расширением Postgres Pro.

См. также

CREATE DOMAIN, ALTER DOMAIN

DROP EVENT TRIGGER

DROP EVENT TRIGGER — удалить событийный триггер

Синтаксис

```
DROP EVENT TRIGGER [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP EVENT TRIGGER` удаляет существующий событийный триггер. Пользователь, выполняющий эту команду, должен быть владельцем событийного триггера.

Параметры

`IF EXISTS`

Не считать ошибкой, если событийный триггер не существует. В этом случае будет выдано замечание.

имя

Имя событийного триггера, подлежащего удалению.

`CASCADE`

Автоматически удалять объекты, зависящие от данного триггера, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении триггера, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление триггера `snitch`:

```
DROP EVENT TRIGGER snitch;
```

Совместимость

Оператор `DROP EVENT TRIGGER` отсутствует в стандарте SQL.

См. также

CREATE EVENT TRIGGER, ALTER EVENT TRIGGER

DROP EXTENSION

DROP EXTENSION — удалить расширение

Синтаксис

```
DROP EXTENSION [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP EXTENSION` удаляет расширения из базы данных. При удалении расширения также удаляются все составляющие его объекты.

Чтобы выполнить `DROP EXTENSION`, необходимо быть владельцем расширения.

Параметры

`IF EXISTS`

Не считать ошибкой, если расширение не существует. В этом случае будет выдано замечание.

имя

Имя установленного расширения.

`CASCADE`

Автоматически удалять объекты, зависящие от данного расширения, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении расширения, если от него зависят какие-либо объекты (кроме объектов, составляющих его, и других расширений, перечисленных в той же команде `DROP`). Это поведение по умолчанию.

Примеры

Удаление расширения `hstore` из текущей базы данных.

```
DROP EXTENSION hstore;
```

Эта команда не будет выполнена, если какие-либо объекты из `hstore` будут задействованы, например, если в какой-либо таблице окажется столбец типа `hstore`. Чтобы принудительно удалить и эти зависимые объекты, необходимо добавить параметр `CASCADE`.

Совместимость

`DROP EXTENSION` является расширением Postgres Pro.

См. также

CREATE EXTENSION, ALTER EXTENSION

DROP FOREIGN DATA WRAPPER

DROP FOREIGN DATA WRAPPER — удалить обёртку сторонних данных

Синтаксис

```
DROP FOREIGN DATA WRAPPER [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP FOREIGN DATA WRAPPER` удаляет существующую обёртку сторонних данных. Пользователь, выполняющий эту команду, должен быть владельцем обёртки.

Параметры

`IF EXISTS`

Не считать ошибкой, если обёртка сторонних данных не существует. В этом случае будет выдано замечание.

имя

Имя существующей обёртки сторонних данных.

`CASCADE`

Автоматически удалять объекты, зависящие от данной обёртки сторонних данных (например, сторонние таблицы и серверы), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении обёртки сторонних данных, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление обёртки сторонних данных `dbi`:

```
DROP FOREIGN DATA WRAPPER dbi;
```

Совместимость

`DROP FOREIGN DATA WRAPPER` соответствует стандарту ISO/IEC 9075-9 (SQL/MED). Предложение `IF EXISTS` является расширением Postgres Pro.

См. также

CREATE FOREIGN DATA WRAPPER, ALTER FOREIGN DATA WRAPPER

DROP FOREIGN TABLE

DROP FOREIGN TABLE — удалить стороннюю таблицу

Синтаксис

```
DROP FOREIGN TABLE [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP FOREIGN TABLE` удаляет стороннюю таблицу. Удалить стороннюю таблицу может только её владелец.

Параметры

`IF EXISTS`

Не считать ошибкой, если сторонняя таблица не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) сторонней таблицы, подлежащей удалению.

`CASCADE`

Автоматически удалять объекты, зависящие от данной сторонней таблицы (например, представления), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении сторонней таблицы, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление двух сторонних таблиц, `films` и `distributors`:

```
DROP FOREIGN TABLE films, distributors;
```

Совместимость

Эта команда соответствует ISO/IEC 9075-9 (SQL/MED), но возможность удалять в одной команде несколько таблиц и указание `IF EXISTS` являются расширениями Postgres Pro.

См. также

ALTER FOREIGN TABLE, CREATE FOREIGN TABLE

DROP FUNCTION

DROP FUNCTION — удалить функцию

Синтаксис

```
DROP FUNCTION [ IF EXISTS ] имя [ ( [ [ режим_аргумента ] [ имя_аргумента
 ] тип_аргумента [, ...] ] ) ] [, ...]
    [ CASCADE | RESTRICT ]
```

Описание

`DROP FUNCTION` удаляет определение существующей функции. Пользователь, выполняющий эту команду, должен быть владельцем функции. Помимо имени функции требуется указать типы её аргументов, так как в базе данных могут существовать несколько функций с одним именем, но с разными списками аргументов.

Параметры

`IF EXISTS`

Не считать ошибкой, если функция не существует. В этом случае будет выдано замечание.

имя

Имя существующей функции (возможно, дополненное схемой). Если список аргументов не указан, имя функции должно быть уникальным в её схеме.

режим_аргумента

Режим аргумента: `IN`, `OUT`, `INOUT` или `VARIADIC`. По умолчанию подразумевается `IN`. Заметьте, что `DROP FUNCTION` не учитывает аргументы `OUT`, так как для идентификации функции нужны только типы входных аргументов. Поэтому достаточно перечислить только аргументы `IN`, `INOUT` и `VARIADIC`.

имя_аргумента

Имя аргумента. Заметьте, что на самом деле `DROP FUNCTION` не обращает внимание на имена аргументов, так как для однозначной идентификации функции достаточно только типов аргументов.

тип_аргумента

Тип данных аргументов функции (возможно, дополненный именем схемы), если таковые имеются.

`CASCADE`

Автоматически удалять объекты, зависящие от данной функции (например, операторы или триггеры), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении функции, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Эта команда удаляет функцию, вычисляющую квадратный корень:

```
DROP FUNCTION sqrt(integer);
```

Удаление нескольких функций одной командой:

```
DROP FUNCTION sqrt(integer), sqrt(bigint);
```

Если имя функции уникально в её схеме, на неё можно сослаться без списка аргументов:

```
DROP FUNCTION update_employee_salaries;
```

Заметьте, что это отличается от

```
DROP FUNCTION update_employee_salaries();
```

Данная форма ссылается на функцию с нулём аргументов, тогда как первый вариант подходит для функции с любым числом аргументов, в том числе и с нулём, если имя функции уникально.

Совместимость

Эта команда соответствует стандарту SQL, но дополнена следующими расширениями Postgres Pro:

- Стандарт допускает удаление в одной команде только одной функции.
- Указание `IF EXISTS`
- Возможность указывать режимы и имена аргументов

См. также

CREATE FUNCTION, ALTER FUNCTION

DROP GROUP

DROP GROUP — удалить роль в базе данных

Синтаксис

```
DROP GROUP [ IF EXISTS ] имя [, ...]
```

Описание

`DROP GROUP` теперь является синонимом оператора DROP ROLE.

Совместимость

Оператор `DROP GROUP` отсутствует в стандарте SQL.

См. также

DROP ROLE

DROP INDEX

DROP INDEX — удалить индекс

Синтаксис

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP INDEX` удаляет существующий индекс из базы данных. Выполнить эту команду может только владелец индекса.

Параметры

`CONCURRENTLY`

С этим указанием индекс удаляется, не блокируя одновременные операции выборки, добавления, изменения и удаления данных в таблице индекса. Обычный оператор `DROP INDEX` запрашивает блокировку `ACCESS EXCLUSIVE` для таблицы, не допуская другие обращения к ней до завершения удаления. Если же добавлено это указание, команда, напротив, будет ждать завершения конфликтующих транзакций.

Применяя это указание, надо учитывать несколько особенностей. В частности, при этом можно задать имя только одного индекса, а параметр `CASCADE` не поддерживается. (Таким образом, индекс, поддерживающий ограничение `UNIQUE` или `PRIMARY KEY`, так удалить нельзя.) Кроме того, обычную команду `DROP INDEX` можно выполнить в блоке транзакции, а `DROP INDEX CONCURRENTLY` — нет.

Для временных таблиц `DROP INDEX` всегда выполняется более простым, неблокирующим способом, так как они не могут использоваться никакими другими сеансами.

`IF EXISTS`

Не считать ошибкой, если индекс не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) индекса, подлежащего удалению.

`CASCADE`

Автоматически удалять объекты, зависящие от данного индекса, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении индекса, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Эта команда удалит индекс `title_idx`:

```
DROP INDEX title_idx;
```

Совместимость

`DROP INDEX` является языковым расширением Postgres Pro. Средства обеспечения индексов в стандарте SQL не описаны.

См. также

CREATE INDEX

DROP LANGUAGE

DROP LANGUAGE — удалить процедурный язык

Синтаксис

```
DROP [ PROCEDURAL ] LANGUAGE [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP LANGUAGE` удаляет определение ранее зарегистрированного процедурного языка. Выполнить `DROP LANGUAGE` может только владелец языка или суперпользователь.

> **Примечание**
>
> Начиная с PostgreSQL 9.1, большинство процедурных языков были преобразованы в «расширения», так что теперь их следует удалять командами DROP EXTENSION, а не `DROP LANGUAGE`.

Параметры

`IF EXISTS`

Не считать ошибкой, если язык не существует. В этом случае будет выдано замечание.

имя

Имя существующего процедурного языка. В целях обратной совместимости имя можно заключить в апострофы.

`CASCADE`

Автоматически удалять объекты, зависящие от данного языка (например, функции на этом языке), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении языка, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Эта команда удаляет процедурный язык `plsample`:

```
DROP LANGUAGE plsample;
```

Совместимость

Оператор `DROP LANGUAGE` отсутствует в стандарте SQL.

См. также

ALTER LANGUAGE, CREATE LANGUAGE

DROP MATERIALIZED VIEW

DROP MATERIALIZED VIEW — удалить материализованное представление

Синтаксис

```
DROP MATERIALIZED VIEW [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP MATERIALIZED VIEW` удаляет существующее материализованное представление. Выполнить эту команду может только владелец материализованного представления.

Параметры

`IF EXISTS`

Не считать ошибкой, если материализованное представление не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) материализованного представления, подлежащего удалению.

`CASCADE`

Автоматически удалять объекты, зависящие от данного материализованного представления (например, другие материализованные или обычные представления), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении материализованного представления, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Эта команда удаляет материализованное представление с именем `order_summary`:

```
DROP MATERIALIZED VIEW order_summary;
```

Совместимость

`DROP MATERIALIZED VIEW` — расширение Postgres Pro.

См. также

CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

DROP OPERATOR

DROP OPERATOR — удалить оператор

Синтаксис

```
DROP OPERATOR [ IF EXISTS ] имя ( { тип_слева | NONE } , { тип_справа | NONE } )
 [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP OPERATOR` удаляет существующий оператор из базы данных. Выполнить эту команду может только владелец оператора.

Параметры

`IF EXISTS`

Не считать ошибкой, если оператор не существует. В этом случае будет выдано замечание.

имя

Имя существующего оператора (возможно, дополненное схемой).

тип_слева

Тип данных левого операнда оператора; если у оператора нет левого операнда, укажите `NONE`.

тип_справа

Тип данных правого операнда оператора; если у оператора нет правого операнда, укажите `NONE`.

`CASCADE`

Автоматически удалять объекты, зависящие от данного оператора (например, использующие его представления), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении оператора, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление оператора возведения в степень `a^b` для типа `integer`:

```
DROP OPERATOR ^ (integer, integer);
```

Удаление левого унарного оператора двоичного дополнения `~b` для типа `bit`:

```
DROP OPERATOR ~ (none, bit);
```

Удаление правого унарного оператора вычисления факториала `x!` для типа `bigint`:

```
DROP OPERATOR ! (bigint, none);
```

Удаление нескольких операторов одной командой:

```
DROP OPERATOR ~ (none, bit), ! (bigint, none);
```

Совместимость

Команда `DROP OPERATOR` отсутствует в стандарте SQL.

См. также

CREATE OPERATOR, ALTER OPERATOR

DROP OPERATOR CLASS

DROP OPERATOR CLASS — удалить класс операторов

Синтаксис

```
DROP OPERATOR CLASS [ IF EXISTS ] имя USING индексный_метод [ CASCADE | RESTRICT ]
```

Описание

`DROP OPERATOR CLASS` удаляет существующий класс операторов. Выполнить эту команду может только владелец класса операторов.

`DROP OPERATOR CLASS` не удаляет операторы или функции, связанные с этим классом. Если же существуют индексы, зависящие от этого класса, класс будет удалён успешно (вместе с индексами), только если добавить указание `CASCADE`.

Параметры

`IF EXISTS`

Не считать ошибкой, если класс операторов не существует. В этом случае будет выдано замечание.

имя

Имя существующего класса операторов (возможно, дополненное схемой).

индексный_метод

Имя индексного метода, для которого предназначен этот класс операторов.

`CASCADE`

Автоматически удалять объекты, зависящие от данного класса операторов (например, использующие его индексы), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении класса операторов, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Замечания

`DROP OPERATOR CLASS` не удалит семейство операторов, содержавшее этот класс, даже если в этом семействе больше ничего не останется (в том числе, если семейство было неявно создано командой `CREATE OPERATOR CLASS`). Пустое семейство операторов безвредно, но порядка ради затем следует удалить и его, командой `DROP OPERATOR FAMILY`; или, возможно, выполнить `DROP OPERATOR FAMILY` в первую очередь.

Примеры

Удаление класса операторов B-дерева с именем `widget_ops`:

```
DROP OPERATOR CLASS widget_ops USING btree;
```

Эта команда не будет выполнена, если в базе существуют индексы, использующие этот класс. Чтобы удалить такие индексы вместе с классом операторов, нужно добавить указание `CASCADE`.

Совместимость

Команда `DROP OPERATOR CLASS` отсутствует в стандарте SQL.

См. также

ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR FAMILY

DROP OPERATOR FAMILY

DROP OPERATOR FAMILY — удалить семейство операторов

Синтаксис

```
DROP OPERATOR FAMILY [ IF EXISTS ] имя USING индексный_метод [ CASCADE | RESTRICT ]
```

Описание

`DROP OPERATOR FAMILY` удаляет существующее семейство операторов. Выполнить эту команду может только владелец семейства операторов.

`DROP OPERATOR FAMILY` удаляет также все классы операторов, содержащиеся в семействе, но не удаляет связанные с ним операторы или функции. Если от классов операторов, содержащихся в семействе, зависят какие-либо индексы, семейство будет удалено успешно (вместе с классами и индексами), только если добавить указание `CASCADE`.

Параметры

`IF EXISTS`

Не считать ошибкой, если семейство операторов не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) существующего семейства операторов.

индексный_метод

Имя индексного метода, для которого предназначено это семейство операторов.

`CASCADE`

Автоматически удалять объекты, зависящие от данного семейства операторов, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении семейства операторов, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление семейства операторов B-дерева с именем `float_ops`:

```
DROP OPERATOR FAMILY float_ops USING btree;
```

Эта команда не будет выполнена, если в базе существуют индексы, использующие классы операторов из этого семейства. Чтобы удалить такие индексы вместе с семейством операторов, нужно добавить указание `CASCADE`.

Совместимость

Команда `DROP OPERATOR FAMILY` отсутствует в стандарте SQL.

См. также

ALTER OPERATOR FAMILY, CREATE OPERATOR FAMILY, ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR CLASS

DROP OWNED

DROP OWNED — удалить объекты базы данных, принадлежащие роли

Синтаксис

```
DROP OWNED BY { имя | CURRENT_USER | SESSION_USER } [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP OWNED` удаляет в текущей базе данных все объекты, принадлежащие любой из указанных ролей. Кроме того, эти роли лишаются всех прав, которые они имели для объектов текущей базы данных или общих объектов (баз данных, табличных пространств).

Параметры

имя

Имя роли, все объекты которой будут уничтожены, а права отозваны.

`CASCADE`

Автоматически удалять объекты, зависящие от каждого затрагиваемого объекта, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении объектов, принадлежащих роли, если от каких-либо из них зависят другие объекты в базе данных. Это поведение по умолчанию.

Замечания

`DROP OWNED` часто применяется при подготовке к удалению одной или нескольких ролей. Так как команда `DROP OWNED` затрагивает объекты только в текущей базе данных, обычно её нужно выполнять в каждой базе данных, которая содержит объекты, принадлежащие удаляемой роли.

С указанием `CASCADE` эта команда может рекурсивно удалить объекты, принадлежащие и другим пользователям.

Команда REASSIGN OWNED даёт альтернативную возможность — сменить владельца для всех объектов базы, принадлежащих одной или нескольким ролям. Однако `REASSIGN OWNED` не затрагивает права, назначенные для объектов, не принадлежащих указанным ролям.

Базы данных и табличные пространства, принадлежащие указанным ролям, эта команда не удаляет.

За подробностями обратитесь к Разделу 20.4.

Совместимость

Команда `DROP OWNED` — расширение Postgres Pro.

См. также

REASSIGN OWNED, DROP ROLE

DROP POLICY

DROP POLICY — удалить политику защиты на уровне строк для таблицы

Синтаксис

```
DROP POLICY [ IF EXISTS ] имя ON имя_таблицы [ CASCADE | RESTRICT ]
```

Описание

`DROP POLICY` удаляет указанную политику из таблицы. Заметьте, что если из таблицы удаляется последняя политика, а в таблице продолжает действовать защита на уровне строк (включённая командой `ALTER TABLE`), будет применена политика запрета по умолчанию. Отключить защиту на уровне строк для таблицы можно с помощью команды `ALTER TABLE ... DISABLE ROW LEVEL SECURITY`, независимо от того, определены ли политики для этой таблицы или нет.

Параметры

`IF EXISTS`

Не считать ошибкой, если политика не существует. В этом случае будет выдано замечание.

имя

Имя политики, подлежащей удалению.

имя_таблицы

Имя (возможно, дополненное схемой) таблицы, к которой применяется эта политика.

`CASCADE`
`RESTRICT`

Эти ключевые слова игнорируются, так как от политик не зависят никакие объекты.

Примеры

Удаление политики `p1` из таблицы `my_table`:

```
DROP POLICY p1 ON my_table;
```

Совместимость

`DROP POLICY` является расширением Postgres Pro.

См. также

CREATE POLICY, ALTER POLICY

DROP PUBLICATION

DROP PUBLICATION — удалить публикацию

Синтаксис

```
DROP PUBLICATION [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP PUBLICATION` удаляет существующую публикацию из базы данных.

Удалить публикацию может только её владелец или суперпользователь.

Параметры

`IF EXISTS`

Не считать ошибкой, если публикация не существует. В этом случае будет выдано замечание.

имя

Имя существующей публикации.

`CASCADE`
`RESTRICT`

Эти ключевые слова игнорируются, так как от публикаций не зависят никакие объекты.

Примеры

Удаление публикации:

```
DROP PUBLICATION mypublication;
```

Совместимость

`DROP PUBLICATION` является расширением Postgres Pro.

См. также

CREATE PUBLICATION, ALTER PUBLICATION

DROP ROLE

DROP ROLE — удалить роль в базе данных

Синтаксис

```
DROP ROLE [ IF EXISTS ] имя [, ...]
```

Описание

`DROP ROLE` удаляет указанные роли. Удалить роль суперпользователя может только суперпользователь, а чтобы удалить роль обычного пользователя, достаточно иметь право `CREATEROLE`.

Если на эту роль есть ссылки в какой-либо базе данных в кластере, возникнет ошибка и роль не будет удалена. Прежде чем удалять роль, необходимо удалить все принадлежащие ей объекты (или сменить их владельца), а также лишить её данных ей прав для других объектов. Для этой цели можно применить команды REASSIGN OWNED и DROP OWNED; за подробностями обратитесь к Разделу 20.4.

Однако ликвидировать членство в ролях, связанное с этой ролью, не требуется; `DROP ROLE` автоматически исключит данную роль из других ролей, и третьи роли из данной. Сами роли при этом не удаляются и другим образом никак не затрагиваются.

Параметры

`IF EXISTS`

Не считать ошибкой, если роль не существует. В этом случае будет выдано замечание.

имя

Имя роли, подлежащей удалению.

Замечания

Postgres Pro включает программу dropuser, которая предоставляет ту же функциональность (на самом деле она вызывает эту команду), но может запускаться в командной оболочке.

Примеры

Удаление роли:

```
DROP ROLE jonathan;
```

Совместимость

В стандарте SQL определена команда `DROP ROLE`, но она может удалять только по одной роли, а для её выполнения требуются другие права, не такие как в Postgres Pro.

См. также

CREATE ROLE, ALTER ROLE, SET ROLE

DROP RULE

DROP RULE — удалить правило перезаписи

Синтаксис

```
DROP RULE [ IF EXISTS ] имя ON имя_таблицы [ CASCADE | RESTRICT ]
```

Описание

`DROP RULE` удаляет правило перезаписи.

Параметры

`IF EXISTS`

Не считать ошибкой, если правило не существует. В этом случае будет выдано замечание.

имя

Имя правила, подлежащего удалению.

имя_таблицы

Имя (возможно, дополненное схемой) существующей таблицы (или представления), к которой применяется это правило.

`CASCADE`

Автоматически удалять объекты, зависящие от данного правила, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении правила, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление правила перезаписи `newrule`:

```
DROP RULE newrule ON mytable;
```

Совместимость

`DROP RULE` является языковым расширением Postgres Pro, как и вся система перезаписи запросов.

См. также

CREATE RULE, ALTER RULE

DROP SCHEMA

DROP SCHEMA — удалить схему

Синтаксис

```
DROP SCHEMA [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP SCHEMA` удаляет схемы из базы данных.

Схему может удалить только её владелец или суперпользователь. Заметьте, что владелец может удалить схему (вместе со всеми содержащимися в ней объектами), даже если он не владеет некоторыми объектами в своей схеме.

Параметры

`IF EXISTS`

Не считать ошибкой, если схема не существует. В этом случае будет выдано замечание.

имя

Имя схемы.

`CASCADE`

Автоматически удалять объекты, содержащиеся в этой схеме (таблицы, функции и т. д.), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении схемы, если она содержит какие-либо объекты. Это поведение по умолчанию.

Замечания

С указанием `CASCADE` эта команда может удалить объекты не только в данной схеме, но и в других.

Примеры

Удаление схемы `mystuff` из базы данных вместе со всем, что в ней содержится:

```
DROP SCHEMA mystuff CASCADE;
```

Совместимость

Команда `DROP SCHEMA` полностью соответствует стандарту SQL, но возможность удалять в одной команде несколько схем и указание `IF EXISTS` являются расширениями Postgres Pro.

См. также

ALTER SCHEMA, CREATE SCHEMA

DROP SEQUENCE

DROP SEQUENCE — удалить последовательность

Синтаксис

```
DROP SEQUENCE [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP SEQUENCE` удаляет генераторы числовых последовательностей. Удалить последовательность может только её владелец или суперпользователь.

Параметры

`IF EXISTS`

Не считать ошибкой, если последовательность не существует. В этом случае будет выдано замечание.

имя

Имя последовательности (возможно, дополненное схемой).

`CASCADE`

Автоматически удалять объекты, зависящие от данной последовательности, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении последовательности, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление последовательности `serial`:

```
DROP SEQUENCE serial;
```

Совместимость

`DROP SEQUENCE` соответствует стандарту SQL, но возможность удалять в одной команде несколько последовательностей и указание `IF EXISTS` являются расширениями Postgres Pro.

См. также

CREATE SEQUENCE, ALTER SEQUENCE

DROP SERVER

DROP SERVER — удалить описание стороннего сервера

Синтаксис

```
DROP SERVER [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP SERVER` удаляет существующее описание стороннего сервера. Пользователь, выполняющий эту команду, должен быть владельцем сервера.

Параметры

`IF EXISTS`

Не считать ошибкой, если сервер не существует. В этом случае будет выдано замечание.

имя

Имя существующего сервера.

`CASCADE`

Автоматически удалять объекты, зависящие от данного триггера, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении сервера, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление определения сервера `foo`, если оно существует:

```
DROP SERVER IF EXISTS foo;
```

Совместимость

`DROP SERVER` соответствует стандарту ISO/IEC 9075-9 (SQL/MED). Предложение `IF EXISTS` является расширением Postgres Pro.

См. также

CREATE SERVER, ALTER SERVER

DROP STATISTICS

DROP STATISTICS — удалить расширенную статистику

Синтаксис

```
DROP STATISTICS [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP STATISTICS` удаляет объект(ы) статистики из базы данных. Удалить объект статистики может только владелец объекта, владелец схемы или суперпользователь.

Параметры

`IF EXISTS`

Не считать ошибкой, если объект статистики не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) объекта статистики, подлежащего удалению.

`CASCADE`
`RESTRICT`

Эти ключевые слова игнорируются, так как от статистики не зависят никакие объекты.

Примеры

Удаление двух объектов статистики в разных схемах (если эти объекты отсутствуют, ошибки не будет):

```
DROP STATISTICS IF EXISTS
    accounting.users_uid_creation,
    public.grants_user_role;
```

Совместимость

Оператор `DROP STATISTICS` отсутствует в стандарте SQL.

См. также

ALTER STATISTICS, CREATE STATISTICS

DROP SUBSCRIPTION

DROP SUBSCRIPTION — удалить подписку

Синтаксис

```
DROP SUBSCRIPTION [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP SUBSCRIPTION` удаляет подписку из кластера баз данных.

Удалить подписку может только суперпользователь.

Команду `DROP SUBSCRIPTION` нельзя выполнять в блоке транзакции, если подписка связана со слотом репликации. (Для освобождения слота можно использовать `ALTER SUBSCRIPTION`.)

Параметры

имя

Имя подписки, подлежащей удалению.

`CASCADE`
`RESTRICT`

Эти ключевые слова игнорируются, так как от подписок не зависят никакие объекты.

Замечания

При удалении подписки, связанной со слотом репликации на удалённом узле (это типичная ситуация), команда `DROP SUBSCRIPTION` подключится к удалённому узлу и попытается удалить слот репликации в ходе этой операции. Это необходимо для освобождения ресурсов, выделенных для подписки на удалённом узле. Если при этом происходит сбой, либо из-за недоступности удалённого узла, либо из-за ошибки при удалении слота репликации, либо вообще из-за его отсутствия, команда `DROP SUBSCRIPTION` прерывается. Для разрешения этой ситуации разорвите связь подписки с этим слотом репликации, выполнив команду `ALTER SUBSCRIPTION ... SET (slot_name = NONE)`. После этого команда `DROP SUBSCRIPTION` не будет пытаться выполнять какие-либо действия на удалённом узле. Заметьте, что если удалённый слот репликации фактически продолжает существовать, его нужно будет удалить вручную; в противном случае для него будет по-прежнему сохраняться WAL, что в конце концов может привести к переполнению диска. См. также Подраздел 30.2.1.

Если подписка связана со слотом репликации, команду `DROP SUBSCRIPTION` нельзя выполнять внутри блока транзакции.

Примеры

Удаление подписки:

```
DROP SUBSCRIPTION mysub;
```

Совместимость

`DROP SUBSCRIPTION` является расширением Postgres Pro.

См. также

CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

DROP TABLE

DROP TABLE — удалить таблицу

Синтаксис

```
DROP TABLE [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP TABLE` удаляет таблицы из базы данных. Удалить таблицу может только её владелец, владелец схемы или суперпользователь. Чтобы опустошить таблицу, не удаляя её саму, вместо этой команды следует использовать DELETE или TRUNCATE.

`DROP TABLE` всегда удаляет все индексы, правила, триггеры и ограничения, существующие в целевой таблице. Однако чтобы удалить таблицу, на которую ссылается представление или ограничение внешнего ключа в другой таблице, необходимо дополнительно указать `CASCADE`. (С указанием `CASCADE` зависимое представление удаляется полностью, тогда как в случае с ограничением внешнего ключа удаляется именно это ограничение, а не вся таблица, к которой оно относится.)

Параметры

`IF EXISTS`

Не считать ошибкой, если таблица не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) таблицы, подлежащей удалению.

`CASCADE`

Автоматически удалять объекты, зависящие от данной таблицы (например, представления), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении таблицы, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление таблиц `films` и `distributors`:

```
DROP TABLE films, distributors;
```

Совместимость

Эта команда соответствует стандарту SQL, но возможность удалять в одной команде несколько таблиц и указание `IF EXISTS` являются расширениями Postgres Pro.

См. также

ALTER TABLE, CREATE TABLE

DROP TABLESPACE

DROP TABLESPACE — удалить табличное пространство

Синтаксис

```
DROP TABLESPACE [ IF EXISTS ] имя
```

Описание

`DROP TABLESPACE` удаляет табличное пространство из системы.

Удалить табличное пространство может только его владелец или суперпользователь. Перед удалением его необходимо очистить от всех объектов базы данных. Даже если в текущей базе данных не будет ни одного объекта, находящегося в этом пространстве, в нём вполне могут оставаться объекты других баз данных. Кроме того, если табличное пространство указано в списке temp_tablespaces любого активного сеанса, команда `DROP` может завершиться ошибкой, если в этом пространстве окажутся временные файлы.

Параметры

`IF EXISTS`

Не считать ошибкой, если табличное пространство не существует. В этом случае будет выдано замечание.

имя

Имя табличного пространства.

Замечания

`DROP TABLESPACE` не может быть выполнена в блоке транзакции.

Примеры

Удаление табличного пространства `mystuff` из системы:

```
DROP TABLESPACE mystuff;
```

Совместимость

`DROP TABLESPACE` является расширением Postgres Pro.

См. также

CREATE TABLESPACE, ALTER TABLESPACE

DROP TEXT SEARCH CONFIGURATION

DROP TEXT SEARCH CONFIGURATION — удалить конфигурацию текстового поиска

Синтаксис

```
DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP TEXT SEARCH CONFIGURATION` удаляет существующую конфигурацию текстового поиска. Выполнить эту команду может только владелец конфигурации.

Параметры

`IF EXISTS`

Не считать ошибкой, если конфигурация текстового поиска не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) существующей конфигурации текстового поиска.

`CASCADE`

Автоматически удалять объекты, зависящие от данной конфигурации текстового поиска, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении конфигурации текстового поиска, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление конфигурации текстового поиска `my_english`:

```
DROP TEXT SEARCH CONFIGURATION my_english;
```

Эта команда не будет выполнена, если какие-либо индексы ссылаются на эту конфигурацию в вызовах `to_tsvector`. Чтобы удалить такие индексы вместе с конфигурацией, следует добавить указание `CASCADE`.

Совместимость

Оператор `DROP TEXT SEARCH CONFIGURATION` отсутствует в стандарте SQL.

См. также

ALTER TEXT SEARCH CONFIGURATION, CREATE TEXT SEARCH CONFIGURATION

DROP TEXT SEARCH DICTIONARY

DROP TEXT SEARCH DICTIONARY — удалить словарь текстового поиска

Синтаксис

```
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP TEXT SEARCH DICTIONARY` удаляет существующий словарь текстового поиска. Выполнить эту команду может только владелец словаря.

Параметры

`IF EXISTS`

Не считать ошибкой, если словарь текстового поиска не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) существующего словаря текстового поиска.

`CASCADE`

Автоматически удалять объекты, зависящие от данного словаря текстового поиска, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении словаря текстового поиска, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удалить словарь текстового поиска `english`:

```
DROP TEXT SEARCH DICTIONARY english;
```

Эта команда не будет выполнена, если существуют конфигурации текстового поиска, использующие этот словарь. Чтобы удалить такие конфигурации вместе со словарём, следует добавить указание `CASCADE`.

Совместимость

Оператор `DROP TEXT SEARCH DICTIONARY` отсутствует в стандарте SQL.

См. также

ALTER TEXT SEARCH DICTIONARY, CREATE TEXT SEARCH DICTIONARY

DROP TEXT SEARCH PARSER

DROP TEXT SEARCH PARSER — удалить анализатор текстового поиска

Синтаксис

```
DROP TEXT SEARCH PARSER [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP TEXT SEARCH PARSER` удаляет существующий анализатор текстового поиска. Выполнить эту команду может только суперпользователь.

Параметры

`IF EXISTS`

Не считать ошибкой, если анализатор текстового поиска не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) существующего анализатора текстового поиска.

`CASCADE`

Автоматически удалять объекты, зависящие от данного анализатора текстового поиска, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении анализатора текстового поиска, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление анализатора текстового поиска `my_parser`:

```
DROP TEXT SEARCH PARSER my_parser;
```

Эта команда не будет выполнена, если существуют конфигурации текстового поиска, использующие это анализатор. Чтобы удалить такие конфигурации вместе с анализатором, следует добавить указание `CASCADE`.

Совместимость

Оператор `DROP TEXT SEARCH PARSER` отсутствует в стандарте SQL.

См. также

ALTER TEXT SEARCH PARSER, CREATE TEXT SEARCH PARSER

DROP TEXT SEARCH TEMPLATE

DROP TEXT SEARCH TEMPLATE — удалить шаблон текстового поиска

Синтаксис

```
DROP TEXT SEARCH TEMPLATE [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP TEXT SEARCH TEMPLATE` удаляет существующий шаблон текстового поиска. Выполнить эту команду может только суперпользователь.

Параметры

`IF EXISTS`

Не считать ошибкой, если шаблон текстового поиска не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) существующего шаблона текстового поиска.

`CASCADE`

Автоматически удалять объекты, зависящие от данного шаблона текстового поиска, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении шаблона текстового поиска, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление шаблона текстового поиска `thesaurus`:

```
DROP TEXT SEARCH TEMPLATE thesaurus;
```

Эта команда не будет выполнена, если существуют словари текстового поиска, использующие этот шаблон. Чтобы удалить такие словари вместе с шаблоном, следует добавить указание `CASCADE`.

Совместимость

Оператор `DROP TEXT SEARCH TEMPLATE` отсутствует в стандарте SQL.

См. также

ALTER TEXT SEARCH TEMPLATE, CREATE TEXT SEARCH TEMPLATE

DROP TRANSFORM

DROP TRANSFORM — удалить трансформацию

Синтаксис

```
DROP TRANSFORM [ IF EXISTS ] FOR имя_типа LANGUAGE имя_языка [ CASCADE | RESTRICT ]
```

Описание

`DROP TRANSFORM` удаляет ранее определённую трансформацию.

Чтобы удалить трансформацию, необходимо быть владельцем типа и языка. Такие же требования действуют и при создании трансформации.

Параметры

`IF EXISTS`

Не считать ошибкой, если трансформация не существует. В этом случае будет выдано замечание.

имя_типа

Имя типа данных, для которого предназначена трансформация.

имя_языка

Имя языка, для которого предназначена трансформация.

`CASCADE`

Автоматически удалять объекты, зависящие от данной трансформации, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении трансформации, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление трансформации для типа `hstore` и языка `plpythonu`:

```
DROP TRANSFORM FOR hstore LANGUAGE plpythonu;
```

Совместимость

Эта форма `DROP TRANSFORM` является расширением Postgres Pro. За подробностями обратитесь к описанию CREATE TRANSFORM.

См. также

CREATE TRANSFORM

DROP TRIGGER

DROP TRIGGER — удалить триггер

Синтаксис

```
DROP TRIGGER [ IF EXISTS ] имя ON имя_таблицы [ CASCADE | RESTRICT ]
```

Описание

`DROP TRIGGER` удаляет существующее определение триггера. Пользователь, выполняющий эту команду, должен быть владельцем таблицы, для которой определён данный триггер.

Параметры

`IF EXISTS`

Не считать ошибкой, если триггер не существует. В этом случае будет выдано замечание.

имя

Имя триггера, подлежащего удалению.

имя_таблицы

Имя (возможно, дополненное схемой) таблицы, для которой определён триггер.

`CASCADE`

Автоматически удалять объекты, зависящие от данного триггера, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении триггера, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление триггера `if_dist_exists` в таблице `films`:

```
DROP TRIGGER if_dist_exists ON films;
```

Совместимость

Оператор `DROP TRIGGER` в Postgres Pro несовместим со стандартом SQL. В стандарте имена триггеров не считаются локальными по отношению к таблицам, так что синтаксис команды проще: `DROP TRIGGER` *имя*.

См. также

CREATE TRIGGER

DROP TYPE

DROP TYPE — удалить тип данных

Синтаксис

```
DROP TYPE [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP TYPE` удаляет определённый пользователем тип данных. Удалить тип может только его владелец.

Параметры

`IF EXISTS`

Не считать ошибкой, если тип не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) типа данных, подлежащего удалению.

`CASCADE`

Автоматически удалять объекты, зависящие от данного типа (например, столбцы таблиц, функции и операторы), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении типа, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление типа данных `box`:

```
DROP TYPE box;
```

Совместимость

Эта команда подобна соответствующей команде в стандарте SQL, но указание `IF EXISTS` является расширением Postgres Pro. Однако учтите, что команда `CREATE TYPE` и механизм расширения типов в Postgres Pro значительно отличаются от стандарта SQL.

См. также

ALTER TYPE, CREATE TYPE

DROP USER

DROP USER — удалить роль в базе данных

Синтаксис

```
DROP USER [ IF EXISTS ] имя [, ...]
```

Описание

`DROP USER` — просто альтернативное написание команды DROP ROLE.

Совместимость

`DROP USER` является расширением Postgres Pro. В стандарте SQL определение пользователей считается зависимым от реализации.

См. также

DROP ROLE

DROP USER MAPPING

DROP USER MAPPING — удалить сопоставление пользователя для стороннего сервера

Синтаксис

```
DROP USER MAPPING [ IF EXISTS ] FOR { имя_пользователя | USER | CURRENT_USER | PUBLIC }
 SERVER имя_сервера
```

Описание

`DROP USER MAPPING` удаляет существующее сопоставление пользователя для стороннего сервера.

Владелец стороннего сервера может удалить заданные для этого сервера сопоставления любых пользователей. Кроме того, обычный пользователь может удалить сопоставление для собственного имени, если он имеет право `USAGE` для сервера.

Параметры

`IF EXISTS`

Не считать ошибкой, если сопоставление не существует. В этом случае будет выдано замечание.

имя_пользователя

Имя пользователя для сопоставления. Значения `CURRENT_USER` и `USER` соответствуют имени текущего пользователя. Значение `PUBLIC` соответствует именам всех текущих и будущих пользователей системы.

имя_сервера

Имя сервера, для которого меняется сопоставление пользователей.

Примеры

Удаление сопоставления пользователя `bob` для сервера `foo`, если оно существует:

```
DROP USER MAPPING IF EXISTS FOR bob SERVER foo;
```

Совместимость

`DROP USER MAPPING` соответствует стандарту ISO/IEC 9075-9 (SQL/MED). Предложение `IF EXISTS` является расширением Postgres Pro.

См. также

CREATE USER MAPPING, ALTER USER MAPPING

DROP VIEW

DROP VIEW — удалить представление

Синтаксис

```
DROP VIEW [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP VIEW` удаляет существующее представление. Выполнить эту команду может только владелец представления.

Параметры

`IF EXISTS`

Не считать ошибкой, если представление не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) представления, подлежащего удалению.

`CASCADE`

Автоматически удалять объекты, зависящие от данного представления (например, другие представления), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.13).

`RESTRICT`

Отказать в удалении представления, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Эта команда удаляет представление с именем `kinds`:

```
DROP VIEW kinds;
```

Совместимость

Эта команда соответствует стандарту SQL, но возможность удалять в одной команде несколько представлений и указание `IF EXISTS` являются расширениями Postgres Pro.

См. также

ALTER VIEW, CREATE VIEW

END

END — зафиксировать текущую транзакцию

Синтаксис

```
END [ WORK | TRANSACTION ]
```

Описание

`END` фиксирует текущую транзакцию. Все изменения, произведённые этой транзакцией, становятся видимыми для других и гарантированно сохранятся в случае сбоя. Эта команда является расширением Postgres Pro и равнозначна команде COMMIT.

Параметры

`WORK`
`TRANSACTION`

Необязательные ключевые слова, не оказывают никакого влияния.

Замечания

Для прерывания транзакции используйте ROLLBACK.

При попытке выполнить `END` вне транзакции ничего не произойдёт, но будет выдано предупреждение.

Примеры

Следующая команда фиксирует текущую транзакцию и сохраняет все изменения:

```
END;
```

Совместимость

`END` является расширением Postgres Pro и выполняет ту же функцию, что и оператор COMMIT, описанный в стандарте SQL.

См. также

BEGIN, COMMIT, ROLLBACK

EXECUTE

EXECUTE — выполнить подготовленный оператор

Синтаксис

```
EXECUTE имя [ ( параметр [, ...] ) ]
```

Описание

`EXECUTE` выполняет подготовленный ранее оператор. Так как подготовленные операторы существуют только в рамках сеанса, они должны создаваться командой `PREPARE`, выполненной в текущем сеансе ранее.

Если команда `PREPARE`, создающая оператор, определяет некоторый набор параметров, команде `EXECUTE` должны быть переданы подходящие значения этих параметров; в противном случае возникнет ошибка. Заметьте, что подготовленные операторы (в отличие от функций) не перегружаются в зависимости от типа или числа параметров; имя подготовленного оператора должно быть уникальным в рамках текущего сеанса.

Чтобы узнать больше о создании и использовании подготовленных операторов, обратитесь к PREPARE.

Параметры

имя

Имя подготовленного оператора, который будет выполнен.

параметр

Фактическое значение параметра подготовленного оператора. Это может быть выражение, выдающее значение, совместимое с типом данных этого параметра, который был определён при создании подготовленного оператора.

Выводимая информация

Метка команды, возвращаемая `EXECUTE`, соответствует подготовленному оператору, а не оператору `EXECUTE`.

Примеры

Примеры приведены в разделе «Примеры» документации по оператору PREPARE.

Совместимость

В стандарте SQL есть оператор `EXECUTE`, но он предназначен только для применения во встраиваемом SQL. Эта версия оператора `EXECUTE` имеет также несколько другой синтаксис.

См. также

DEALLOCATE, PREPARE

EXPLAIN

EXPLAIN — показать план выполнения оператора

Синтаксис

```
EXPLAIN [ ( параметр [, ...] ) ] оператор
EXPLAIN [ ANALYZE ] [ VERBOSE ] оператор

Здесь допускается параметр:

    ANALYZE [ boolean ]
    VERBOSE [ boolean ]
    COSTS [ boolean ]
    BUFFERS [ boolean ]
    TIMING [ boolean ]
    SUMMARY [ boolean ]
    FORMAT { TEXT | XML | JSON | YAML }
```

Описание

Эта команда выводит план выполнения, генерируемый планировщиком Postgres Pro для заданного оператора. План выполнения показывает, как будут сканироваться таблицы, затрагиваемые оператором — просто последовательно, по индексу и т. д. — а если запрос связывает несколько таблиц, какой алгоритм соединения будет выбран для объединения считанных из них строк.

Наибольший интерес в выводимой информации представляет ожидаемая стоимость выполнения оператора, которая показывает, сколько, по мнению планировщика, будет выполняться этот оператор (это значение измеряется в единицах стоимости, которые не имеют точного определения, но обычно это обращение к странице на диске). Фактически выводятся два числа: стоимость запуска до выдачи первой строки и общая стоимость выдачи всех строк. Для большинства запросов важна общая стоимость, но в таких контекстах, как подзапрос в EXISTS, планировщик будет минимизировать стоимость запуска, а не общую стоимость (так как исполнение запроса всё равно завершится сразу после получения одной строки). Кроме того, если количество возвращаемых строк ограничивается предложением LIMIT, планировщик интерполирует стоимость между двумя этими числами, выбирая наиболее выгодный план.

С параметром ANALYZE оператор будет выполнен на самом деле, а не только запланирован. При этом в вывод добавляются фактические сведения о времени выполнения, включая общее время, затраченное на каждый узел плана (в миллисекундах) и общее число строк, выданных в результате. Это помогает понять, насколько близки к реальности предварительные оценки планировщика.

> **Важно**
>
> Имейте в виду, что с указанием ANALYZE оператор действительно выполняется. Хотя EXPLAIN отбрасывает результат, который вернул бы SELECT, в остальном все действия выполняются как обычно. Если вы хотите выполнить EXPLAIN ANALYZE с командой INSERT, UPDATE, DELETE, CREATE TABLE AS или EXECUTE, не допуская изменения данных этой командой, воспользуйтесь таким приёмом:
>
> ```
> BEGIN;
> EXPLAIN ANALYZE ...;
> ROLLBACK;
> ```

Без скобок для этого оператора можно указать только параметры ANALYZE и VERBOSE и только в таком порядке. В PostgreSQL до версии 9.0 поддерживался только синтаксис без скобок, однако в дальнейшем ожидается, что все новые параметры будут восприниматься только в скобках.

Параметры

`ANALYZE`

Выполнить команду и вывести фактическое время выполнения и другую статистику. По умолчанию этот параметр равен `FALSE`.

`VERBOSE`

Вывести дополнительную информацию о плане запроса. В частности, включить список столбцов результата для каждого узла в дереве плана, дополнить схемой имена таблиц и функций, всегда указывать для переменных в выражениях псевдоним их таблицы, а также выводить имена всех триггеров, для которых выдаётся статистика. По умолчанию этот параметр равен `FALSE`.

`COSTS`

Вывести рассчитанную стоимость запуска и общую стоимость каждого узла плана, а также рассчитанное число строк и ширину каждой строки. Этот параметр по умолчанию равен `TRUE`.

`BUFFERS`

Включить информацию об использовании буфера. В частности, вывести число попаданий, блоков прочитанных, загрязнённых и записанных в разделяемом и локальном буфере, а также число прочитанных и записанных временных блоков. *Попаданием* (hit) считается ситуация, когда требуемый блок уже находится в кеше и чтения с диска удаётся избежать. Блоки в общем буфере содержат данные обычных таблиц и индексов, в локальном — данные временных таблиц и индексов, а временные блоки предназначены для краткосрочного использования при выполнении сортировки, хеширования, материализации и подобных узлов плана. Число *загрязнённых* блоков (dirtied) показывает, сколько ранее не модифицированных блоков изменила данная операция; тогда как число *записанных* блоков (written) показывает, сколько ранее загрязнённых блоков данный серверный процесс вынес из кеша при обработке запроса. Значения, указываемые для узла верхнего уровня, включают значения всех его дочерних узлов. В текстовом формате выводятся только ненулевые значения. Этот параметр действует только в режиме `ANALYZE`. По умолчанию его значение равно `FALSE`.

`TIMING`

Включить в вывод фактическое время запуска и время, затраченное на каждый узел. Постоянное чтение системных часов может значительно замедлить запрос, так что если достаточно знать фактическое число строк, имеет смысл сделать этот параметр равным `FALSE`. Время выполнения всего оператора замеряется всегда, даже когда этот параметр выключен и на уровне узлов время не подсчитывается. Этот параметр действует только в режиме `ANALYZE`. По умолчанию его значение равно `TRUE`.

`SUMMARY`

Включить сводку (например, суммарное время) после плана запроса. Сводка включается по умолчанию, когда используется `ANALYZE`, но этот параметр позволяет получить её и с другими вариантами команды. Время планирования в `EXPLAIN EXECUTE` включает время извлечения плана из кеша и время перепланирования, если оно потребовалось.

`FORMAT`

Установить один из следующих форматов вывода: TEXT, XML, JSON или YAML. Последние три формата содержат ту же информацию, что и текстовый, но больше подходят для программного разбора. По умолчанию выбирается формат `TEXT`.

`boolean`

Включает или отключает заданный параметр. Для включения параметра можно написать `TRUE`, `ON` или `1`, а для отключения — `FALSE`, `OFF` или `0`. Значение *`boolean`* можно опустить, в этом случае подразумевается `TRUE`.

оператор

Любой оператор SELECT, INSERT, UPDATE, DELETE, VALUES, EXECUTE, DECLARE, CREATE TABLE AS и CREATE MATERIALIZED VIEW AS, план выполнения которого вас интересует.

Выводимая информация

Результатом команды будет текстовое описание плана, выбранного для *оператора*, возможно, дополненное статистикой выполнения. Представленная информация описана в Разделе 14.1.

Замечания

Чтобы планировщик запросов Postgres Pro был достаточно информирован для эффективной оптимизации запросов, данные в pg_statistic должны быть актуальными для всех таблиц, задействованных в запросе. Обычно об этом автоматически заботится демон автоочистки. Но если в таблице недавно произошли значительные изменения, может потребоваться вручную выполнить ANALYZE, не дожидаясь, пока автоочистка обработает эти изменения.

Измеряя фактическую стоимость выполнения каждого узла в плане, текущая реализация EXPLAIN ANALYZE привносит накладные расходы профилирования в выполнение запроса. В результате этого, при запуске запроса командой EXPLAIN ANALYZE он может выполняться значительно дольше, чем при обычном выполнении. Объём накладных расходов зависит от природы запроса, а также от используемой платформы. Худшая ситуация наблюдается для узлов плана, которые сами по себе выполняются очень быстро, и в операционных системах, где получение текущего времени относительно длительная операция.

Примеры

Получение плана простого запроса для таблицы, содержащей единственный столбец типа integer, с 10000 строк:

```
EXPLAIN SELECT * FROM foo;

                       QUERY PLAN
-------------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

План того же запроса, но выведенный в формате JSON:

```
EXPLAIN (FORMAT JSON) SELECT * FROM foo;
           QUERY PLAN
--------------------------------
 [                             +
   {                           +
     "Plan": {                 +
       "Node Type": "Seq Scan",+
       "Relation Name": "foo", +
       "Alias": "foo",         +
       "Startup Cost": 0.00,   +
       "Total Cost": 155.00,   +
       "Plan Rows": 10000,     +
       "Plan Width": 4         +
     }                         +
   }                           +
 ]
(1 row)
```

Если в таблице есть индекс, а в запросе присутствует условие WHERE, для которого полезен этот индекс, EXPLAIN может показать другой план:

```
EXPLAIN SELECT * FROM foo WHERE i = 4;

                           QUERY PLAN
-----------------------------------------------------------------
 Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
   Index Cond: (i = 4)
(2 rows)
```

План того же запроса, но в формате YAML:

```
EXPLAIN (FORMAT YAML) SELECT * FROM foo WHERE i='4';
          QUERY PLAN
------------------------------
 - Plan:                      +
     Node Type: "Index Scan"  +
     Scan Direction: "Forward"+
     Index Name: "fi"         +
     Relation Name: "foo"     +
     Alias: "foo"             +
     Startup Cost: 0.00       +
     Total Cost: 5.98         +
     Plan Rows: 1             +
     Plan Width: 4            +
     Index Cond: "(i = 4)"
(1 row)
```

Рассмотрение формата XML оставлено в качестве упражнения для читателя.

План того же запроса без вывода оценок стоимости:

```
EXPLAIN (COSTS FALSE) SELECT * FROM foo WHERE i = 4;

        QUERY PLAN
---------------------------
 Index Scan using fi on foo
   Index Cond: (i = 4)
(2 rows)
```

Пример плана для запроса с агрегатной функцией:

```
EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;

                              QUERY PLAN
---------------------------------------------------------------------
 Aggregate  (cost=23.93..23.93 rows=1 width=4)
   ->  Index Scan using fi on foo  (cost=0.00..23.92 rows=6 width=4)
         Index Cond: (i < 10)
(3 rows)
```

Пример использования `EXPLAIN EXECUTE` для отображения плана выполнения подготовленного запроса:

```
PREPARE query(int, int) AS SELECT sum(bar) FROM test
    WHERE id > $1 AND id < $2
    GROUP BY foo;

EXPLAIN ANALYZE EXECUTE query(100, 200);

                                                        QUERY PLAN

-----------------------------------------------------------------------------------------------------
```

```
 HashAggregate  (cost=9.54..9.54 rows=1 width=8) (actual time=0.156..0.161 rows=11
 loops=1)
   Group Key: foo
   ->  Index Scan using test_pkey on test  (cost=0.29..9.29 rows=50 width=8) (actual
 time=0.039..0.091 rows=99 loops=1)
         Index Cond: ((id > $1) AND (id < $2))
 Planning time: 0.197 ms
 Execution time: 0.225 ms
(6 rows)
```

Разумеется, конкретные числа, показанные здесь, зависят от фактического содержимого задействованных таблиц. Также учтите, что эти числа и даже выбранная стратегия выполнения запроса могут меняться от версии к версии Postgres Pro вследствие усовершенствования планировщика. Кроме того, команда `ANALYZE` при обработке статистических данных производит случайные выборки, так что оценки стоимости могут меняться при каждом чистом запуске `ANALYZE`, даже когда фактическое распределение данных в таблице не меняется.

Совместимость

Оператор `EXPLAIN` отсутствует в стандарте SQL.

См. также

ANALYZE

FETCH

FETCH — получить результат запроса через курсор

Синтаксис

```
FETCH [ направление ] [ FROM | IN ] имя_курсора

Здесь допускается направление:

    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE число
    RELATIVE число
    число
    ALL
    FORWARD
    FORWARD число
    FORWARD ALL
    BACKWARD
    BACKWARD число
    BACKWARD ALL
```

Описание

`FETCH` получает строки через ранее созданный курсор.

Курсор связан с определённым положением, что и использует команда `FETCH`. Курсор может располагаться перед первой строкой результата запроса, на любой строке этого результата, либо после последней строки. При создании курсор оказывается перед первой строкой. Когда `FETCH` доходит до конца набора строк, курсор остаётся в положении после последней строки, либо перед первой, при движении назад. После команд `FETCH ALL` и `FETCH BACKWARD ALL` курсор всегда оказывается после последней строки или перед первой, соответственно.

Формы `NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE` и `RELATIVE` выбирают одну строку после соответствущего перемещения курсора. Если в этом положении строки не оказывается, возвращается пустой результат, а курсор остаётся в достигнутом положении перед первой строкой или после последней.

Формы `FORWARD` и `BACKWARD` получают указанное число строк, сдвигаясь соответственно вперёд или назад; в результате курсор оказывается на последней выданной строке (или перед/после всех строк, если *число* превышает количество доступных строк).

Формы `RELATIVE 0`, `FORWARD 0` и `BACKWARD 0` действуют одинаково — они считывают текущую строку, не перемещая курсор, то есть, повторно выбирая строку, выбранную последней. Эта операция будет успешна, только если курсор не расположен до первой или после последней строки; в этом случае строка возвращена не будет.

> **Примечание**
>
> На этой странице описывается применение курсоров на уровне команд SQL. Если вы попытаетесь использовать курсоры внутри функции PL/pgSQL, правила будут другими — см. Подраздел 40.7.3.

Параметры

направление

Параметр *направление* задаёт направление движения и число выбираемых строк. Он может принимать одно из следующих значений:

NEXT

Выбрать следующую строку. Это действие подразумевается по умолчанию, если *направление* опущено.

PRIOR

Выбрать предыдущую строку.

FIRST

Выбрать первую строку запроса (аналогично указанию ABSOLUTE 1).

LAST

Выбрать последнюю строку запроса (аналогично ABSOLUTE −1).

ABSOLUTE *число*

Выбрать строку под номером *число* с начала, либо под номером abs(*число*) с конца, если *число* отрицательно. Если *число* выходит за границы набора строк, курсор размещается перед первой или после последней строки; в частности, с ABSOLUTE 0 курсор оказывается перед первой строкой.

RELATIVE *число*

Выбрать строку под номером *число*, считая со следующей вперёд, либо под номером abs(*число*), считая с предыдущей назад, если *число* отрицательно. RELATIVE 0 повторно считывает текущую строку, если таковая имеется.

число

Выбрать следующее *число* строк (аналогично FORWARD *число*).

ALL

Выбрать все оставшиеся строки (аналогично FORWARD ALL).

FORWARD

Выбрать следующую строку (аналогично NEXT).

FORWARD *число*

Выбрать следующее *число* строк. FORWARD 0 повторно выбирает текущую строку.

FORWARD ALL

Выбрать все оставшиеся строки.

BACKWARD

Выбрать предыдущую строку (аналогично PRIOR).

BACKWARD *число*

Выбрать предыдущее *число* строк (с перемещением назад). BACKWARD 0 повторно выбирает текущую строку.

BACKWARD ALL

Выбрать все предыдущие строки (с перемещением назад).

число

Здесь *число* — целочисленная константа, возможно со знаком, определяющая смещение или количество выбираемых строк. Для вариантов FORWARD и BACKWARD указание отрицательного *числа* равнозначно смене направления FORWARD на BACKWARD и наоборот.

имя_курсора

Имя открытого курсора.

Выводимая информация

При успешном выполнении FETCH возвращает метку команды вида

FETCH *число*

Здесь *count* — количество выбранных строк (может быть и нулевым). Заметьте, что в psql метка команды не выдаётся, так как вместо неё psql выводит выбранные строки.

Замечания

Если перемещение курсора в FETCH не ограничивается вариантами FETCH NEXT или FETCH FORWARD с положительным числом, курсор должен быть объявлен с указанием SCROLL. Для простых запросов Postgres Pro допускает обратное перемещение курсора, объявленного без SCROLL, но на это поведение лучше не рассчитывать. Если курсор объявлен с указанием NO SCROLL, перемещение назад запрещается.

Вариант ABSOLUTE нисколько не быстрее, чем перемещение к требуемой строке с относительным сдвигом: нижележащий механизм всё равно должен прочитать все промежуточные строки. Выборки по абсолютному отрицательному положению ещё хуже: сначала запрос необходимо прочитать до конца и найти последнюю строку, а затем вернуться назад к указанной строке. Однако перемотка к началу запроса (FETCH ABSOLUTE 0) выполняется быстро.

Определить курсор позволяет команда DECLARE, а переместить его, не читая данные, — команда MOVE.

Примеры

Следующий пример демонстрирует перемещение курсора в таблице:

```
BEGIN WORK;

-- Создание курсора:
DECLARE liahona SCROLL CURSOR FOR SELECT * FROM films;

-- Получение первых 5 строк через курсор liahona:
FETCH FORWARD 5 FROM liahona;

 code  |          title          | did | date_prod  |   kind   |  len
-------+-------------------------+-----+------------+----------+-------
 BL101 | The Third Man           | 101 | 1949-12-23 | Drama    | 01:44
 BL102 | The African Queen       | 101 | 1951-08-11 | Romantic | 01:43
 JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantic | 01:25
 P_301 | Vertigo                 | 103 | 1958-11-14 | Action   | 02:08
 P_302 | Becket                  | 103 | 1964-02-03 | Drama    | 02:28

-- Получение предыдущей строки:
FETCH PRIOR FROM liahona;

 code  |  title  | did | date_prod  |  kind  |  len
-------+---------+-----+------------+--------+-------
 P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08
```

```
-- Закрытие курсора и завершение транзакции:
CLOSE liahona;
COMMIT WORK;
```

Совместимость

В стандарте SQL команда `FETCH` определена только для встраиваемого SQL. Описанная здесь реализация `FETCH` возвращает данные подобно оператору `SELECT`, а не помещает их в переменные исполняющей среды. В остальном, `FETCH` полностью прямо-совместима со стандартом SQL.

Формы `FETCH` с `FORWARD` и `BACKWARD`, а также формы `FETCH` *`число`* и `FETCH ALL` (в которых `FORWARD` подразумевается) являются расширениями Postgres Pro.

В стандарте SQL перед именем курсора допускается только указание `FROM`; возможность указать `IN` или опустить оба указания относится к расширениям.

См. также

CLOSE, DECLARE, MOVE

GRANT

GRANT — определить права доступа

Синтаксис

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] имя_таблицы [, ...]
         | ALL TABLES IN SCHEMA имя_схемы [, ...] }
    TO указание_роли [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( имя_столбца [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( имя_столбца [, ...] ) }
    ON [ TABLE ] имя_таблицы [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE имя_последовательности [, ...]
         | ALL SEQUENCES IN SCHEMA имя_схемы [, ...] }
    TO указание_роли [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE имя_бд [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN имя_домена [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER имя_обёртки_сторонних_данных [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER имя_сервера [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTION имя_функции [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ] [, ...]
         | ALL FUNCTIONS IN SCHEMA имя_схемы [, ...] }
    TO указание_роли [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE имя_языка [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT oid_БО [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA имя_схемы [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE табл_пространство [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE имя_типа [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]

GRANT имя_роли [, ...] TO указание_роли [, ...]
    [ WITH ADMIN OPTION ]
    [ GRANTED BY указание_роли ]

Здесь указание_роли:

    [ GROUP ] имя_роли
  | PUBLIC
  | CURRENT_USER
  | SESSION_USER
```

Описание

Команда GRANT имеет две основные разновидности: первая назначает права для доступа к объектам баз данных (таблицам, столбцам, представлениям, сторонним таблицам, последовательностям, базам данных, обёрткам сторонних данных, сторонним серверам, функциям, процедурным языкам, схемам или табличным пространствам), а вторая назначает одни роли членами других. Эти разновидности во многом похожи, но имеют достаточно отличий, чтобы рассматривать их отдельно.

GRANT для объектов баз данных

Эта разновидность команды GRANT даёт одной или нескольким ролям определённые права для доступа к объекту базы данных. Эти права добавляются к списку имеющихся, если роль уже наделена какими-то правами.

Также можно дать роли некоторое право для всех объектов одного типа в одной или нескольких схемах. Эта функциональность в настоящее время поддерживается только для таблиц, последовательностей и функций (но заметьте, что указание ALL TABLES распространяется также на представления и сторонние таблицы).

Ключевое слово PUBLIC означает, что права даются всем ролям, включая те, что могут быть созданы позже. PUBLIC можно воспринимать как неявно определённую группу, в которую входят все роли. Любая конкретная роль получит в сумме все права, данные непосредственно ей и ролям, членом которых она является, а также права, данные роли PUBLIC.

Если указано WITH GRANT OPTION, получатель права, в свою очередь, может давать его другим. Без этого указания распоряжаться своим правом он не сможет. Группе PUBLIC право передачи права дать нельзя.

Нет необходимости явно давать права для доступа к объекту его владельцу (обычно это пользователь, создавший объект), так как по умолчанию он имеет все права. (Однако владелец может лишить себя прав в целях безопасности.)

Право удалять объект или изменять его определение произвольным образом не считается назначаемым; оно неотъемлемо связано с владельцем, так что отозвать это право или дать его кому-то другому нельзя. (Однако похожий эффект можно получить, управляя членством в роли, владеющей объектом; см. ниже.) Владелец также неявно получает право распоряжения всеми правами для своего объекта.

Postgres Pro по умолчанию назначает группе PUBLIC определённые права для некоторых типов объектов. Для таблиц, столбцов, последовательностей, обёрток сторонних данных, сторонних

серверов, больших объектов, схем или табличных пространств PUBLIC по умолчанию никаких прав не имеет. Для других типов объектов PUBLIC имеет следующие права по умолчанию: CONNECT и TEMPORARY (создание временных таблиц) для баз данных; EXECUTE — для функций, USAGE — для языков и типов данных (включая домены). Владелец объекта, конечно же, может отозвать (с помощью REVOKE) как явно назначенные права, так и права по умолчанию. (Для максимальной безопасности команду REVOKE нужно выполнять в транзакции, создающей объект; тогда не образуется окно, в котором другой пользователь смог бы обратиться к объекту.) Кроме того, эти изначально назначаемые права по умолчанию можно изменить, воспользовавшись командой ALTER DEFAULT PRIVILEGES.

Все возможные права перечислены ниже:

SELECT

Позволяет выполнять SELECT для любого столбца или перечисленных столбцов в заданной таблице, представлении или последовательности. Также позволяет выполнять COPY TO. Помимо того, это право требуется для обращения к существующим значениям столбцов в UPDATE или DELETE. Для последовательностей это право позволяет пользоваться функцией currval. Для больших объектов это право позволяет читать содержимое объекта.

INSERT

Позволяет вставлять строки в заданную таблицу с помощью INSERT. Если право ограничивается несколькими столбцами, только их значение можно будет задать в команде INSERT (другие столбцы получат значения по умолчанию). Также позволяет выполнять COPY FROM.

UPDATE

Позволяет изменять (с помощью UPDATE) данные во всех, либо только перечисленных, столбцах в заданной таблице. (На практике для любой нетривиальной команды UPDATE потребуется и право SELECT, так как она должна обратиться к столбцам таблицы, чтобы определить, какие строки подлежат изменению, и/или вычислить новые значения столбцов.) Для SELECT ... FOR UPDATE и SELECT ... FOR SHARE также необходимо это право как минимум для одного столбца, помимо права SELECT. Для последовательностей это право позволяет пользоваться функциями nextval и setval. Для больших объектов это право позволяет записывать данные в объект или обрезать его.

DELETE

Позволяет удалять строки из заданной таблицы с помощью DELETE. (На практике для любой нетривиальной команды DELETE потребуется также право SELECT, так как она должна обратиться к столбцам таблицы, чтобы определить, какие строки подлежат удалению.)

TRUNCATE

Позволяет опустошить заданную таблицу с помощью TRUNCATE.

REFERENCES

Позволяет создавать ограничение внешнего ключа, ссылающееся на определённую таблицу либо на определённые столбцы таблицы. (См. описание CREATE TABLE.)

TRIGGER

Позволяет создавать триггеры в заданной таблице. (См. описание оператора CREATE TRIGGER.)

CREATE

Для баз данных это право позволяет создавать схемы и публикации в заданной базе.

Для схем это право позволяет создавать новые объекты в заданной схеме. Чтобы переименовать существующий объект, необходимо быть его владельцем *и* иметь это право для схемы, содержащей его.

Для табличных пространств это право позволяет создавать таблицы, индексы и временные файлы в заданном табличном пространстве, а также создавать базы данных, для которых это пространство будет основным. (Учтите, что когда это право отзывается, существующие объекты остаются в прежнем расположении.)

`CONNECT`

Позволяет пользователю подключаться к указанной базе данных. Это право проверяется при установлении соединения (в дополнение к условиям, определённым в конфигурации `pg_hba.conf`).

`TEMPORARY`
`TEMP`

Позволяет создавать временные таблицы в заданной базе данных.

`EXECUTE`

Позволяет выполнять заданную функцию и применять любые определённые поверх неё операторы. Это единственный тип прав, применимый к функциям. (Этот синтаксис распространяется и на агрегатные функции.)

`USAGE`

Для процедурных языков это право позволяет создавать функции на заданном языке. Это единственный тип прав, применимый к процедурным языкам.

Для схем это право даёт доступ к объектам, содержащимся в заданной схеме (предполагается, что при этом имеются права, необходимые для доступа к самим объектам). По сути это право позволяет субъекту «просматривать» объекты внутри схемы. Без этого разрешения имена объектов всё же можно будет узнать, например, обратившись к системным таблицам. Кроме того, если отозвать это право, в существующих сеансах могут оказаться операторы, для которых просмотр имён объектов был выполнен ранее, так что это право не позволяет абсолютно надёжно перекрыть доступ к объектам.

Для последовательностей это право позволяет использовать функции `currval` и `nextval`.

Для типов и доменов это право позволяет использовать заданный тип или домен при создании таблиц, функций или других объектов схемы. (Заметьте, что это право не ограничивает общее «использование» типа, например, обращение к значениям типа в запросах. Не имея этого права, субъект лишается только возможности создавать объекты, зависящие от заданного типа. Основное предназначение этого права в том, чтобы ограничить круг пользователей, способных создавать зависимости от типа, которые могут впоследствии помешать владельцу типа изменить его.)

Для обёрток сторонних данных это право позволяет создавать новые серверы, используя заданную обёртку.

Для серверов это право позволяет создавать определения сторонних таблиц, связанные с этим сервером. Субъекты могут также создавать, изменять или удалять сопоставление для собственного имени пользователя, связанное с этим сервером.

`ALL PRIVILEGES`

Даёт целевой роли все права сразу. Ключевое слово `PRIVILEGES` является необязательным в Postgres Pro, хотя в стандарте SQL оно требуется.

Права, требующиеся для других команд, указаны на страницах справки этих команд.

GRANT для ролей

Эта разновидность команды `GRANT` включает роль в члены одной или нескольких других ролей. Членство в ролях играет важную роль, так как права, данные роли, распространяются и на всех её членов.

Получивший членство в роли с указанием WITH ADMIN OPTION сможет, в свою очередь, включать в члены этой роли, а также исключать из неё другие роли. Без этого указания обычные пользователи не могут это делать. Считается, что роль не имеет права WITH ADMIN OPTION для самой себя, но ей позволено управлять своими членами из сеанса, в котором пользователь сеанса соответствует данной роли. Суперпользователи баз данных могут включать или исключать любые роли из любых ролей. Роли с правом CREATEROLE могут управлять членством в любых ролях, кроме ролей суперпользователей.

С указанием GRANTED BY назначение права сохраняется как данное указанной ролью. Это указание в полной мере могут использовать только суперпользователи базы данных, а обычному пользователю оно доступно, только если в этом указании задаётся имя его роли.

В отличие от прав, членство в ролях нельзя назначить группе PUBLIC. Заметьте также, что эта форма команды не принимает избыточное слово GROUP в *указании_роли*.

Замечания

Для лишения субъектов прав доступа применяется команда REVOKE.

Начиная с PostgreSQL версии 8.1, концепции пользователей и групп объединены в единую сущность, названную ролью. Таким образом, теперь нет необходимости добавлять ключевое слово GROUP, чтобы показать, что субъект является группой, а не пользователем. Слово GROUP всё ещё принимается этой командой, но оно лишено смысловой нагрузки.

Пользователь может выполнять SELECT, INSERT и подобные команды со столбцом таблицы, если он имеет такое право для данного столбца или для всей таблицы. Если назначить пользователю требуемое право на уровне таблицы, а затем отозвать его для одного из столбцов, это не даст эффекта, которого можно было бы ожидать: операция с правами на уровне столбцов не затронет право на уровне таблицы.

Если назначить право доступа к объекту (с помощью GRANT) попытается не владелец объекта, команда завершится ошибкой, если пользователь не имеет никаких прав для этого объекта. Если же пользователь имеет какие-то права, команда будет выполняться, но пользователь сможет давать другим только те права, которые даны ему с правом передачи. Формы GRANT ALL PRIVILEGES будут выдавать предупреждение, если у него вовсе нет таких прав, тогда как другие формы будут выдавать предупреждения, если пользователь не имеет прав распоряжаться именно правами, указанными в команде. (В принципе, эти утверждения применимы и к владельцу объекта, но ему разрешено распоряжаться всеми правами, поэтому такие ситуации невозможны.)

Следует отметить, что суперпользователи баз данных могут обращаться к любым объектам, вне зависимости от наличия каких-либо прав. Это сравнимо с привилегиями пользователя root в системе Unix. И так же, как root, роль суперпользователя следует использовать только когда это абсолютно необходимо.

Если суперпользователь решит выполнить команду GRANT или REVOKE, она будет выполнена, как если бы её выполнял владелец заданного объекта. В частности, права, назначенные такой командой, будут представлены как права, назначенные владельцем объекта. (Если так же установить членство в роли, оно будет представлено как назначенное самой ролью.)

GRANT и REVOKE также могут быть выполнены ролью, которая не является владельцем заданного объекта, но является членом роли-владельца, либо членом роли, имеющей права WITH GRANT OPTION для этого объекта. В этом случае права будут записаны как назначенные ролью, которая действительно владеет объектом, либо имеет право WITH GRANT OPTION. Например, если таблица t1 принадлежит роли g1, членом которой является u1, то u1 может дать права на использование t1 роли u2, но эти права будут представлены, как назначенные непосредственно ролью g1. Отозвать эти права позже сможет любой член роли g1.

Если роль, выполняющая команду GRANT, получает требуемое право по нескольким путям членства, какая именно роль будет выбрана в качестве назначающей право, не определено. Если

это важно, в таких случаях рекомендуется воспользоваться командой SET ROLE и переключиться на роль, которую хочется видеть в качестве выполняющей GRANT.

При назначении прав для доступа к таблице они автоматически не распространяются на последовательности, используемые этой таблицей, в том числе, на последовательности, связанные со столбцами SERIAL. Права доступа к последовательностям нужно назначать отдельно.

Чтобы получить информацию о существующих правах, назначенных для таблиц и столбцов, воспользуйтесь командой \dp в psql. Например:

```
=> \dp mytable
                                Access privileges
 Schema |  Name   | Type  |   Access privileges   | Column access privileges
--------+---------+-------+-----------------------+--------------------------
 public | mytable | table | miriam=arwdDxt/miriam | col1:
                          : =r/miriam             :   miriam_rw=rw/miriam
                          : admin=arw/miriam
(1 row)
```

Записи, выводимые командой \dp, интерпретируются так:

```
имя_роли=xxxx -- права, назначенные роли
        =xxxx -- права, назначенные PUBLIC

            r -- SELECT ("read", чтение)
            w -- UPDATE ("write", запись)
            a -- INSERT ("append", добавление)
            d -- DELETE
            D -- TRUNCATE
            x -- REFERENCES
            t -- TRIGGER
            X -- EXECUTE
            U -- USAGE
            C -- CREATE
            c -- CONNECT
            T -- TEMPORARY
      arwdDxt -- ALL PRIVILEGES (все права для таблиц; для других объектов другие)
            * -- право передачи заданного права

        /yyyy -- роль, назначившая это право
```

В примере выше показано, что увидит пользователь miriam, если создаст таблицу mytable и выполнит:

```
GRANT SELECT ON mytable TO PUBLIC;
GRANT SELECT, UPDATE, INSERT ON mytable TO admin;
GRANT SELECT (col1), UPDATE (col1) ON mytable TO miriam_rw;
```

Для других объектов есть другие команды \d, которые также позволяют просмотреть назначенные права.

Если столбец «Права доступа» для данного объекта пуст, это значит, что для объекта действуют стандартные права (то есть столбец прав содержит NULL). Права по умолчанию всегда включают все права для владельца и могут также включать некоторые права для PUBLIC в зависимости от типа объекта, как разъяснялось выше. Первая команда GRANT или REVOKE для объекта приводит к проявлению записи прав по умолчанию (например, {miriam=arwdDxt/miriam}), а затем изменяет эту запись в соответствии с заданным запросом. Подобным образом, строки, показанные в столбце «Права доступа к столбцам», выводятся только для столбцов с нестандартными правами доступа. (Заметьте, что в данном контексте под «стандартными правами» всегда подразумевается встроенный набор прав, предопределённый для типа объекта. Если с объектом связан набор прав

по умолчанию, полученный после изменения в результате ALTER DEFAULT PRIVILEGES, изменённые права будут всегда выводиться явно, показывая эффект команды ALTER.)

Заметьте, что право распоряжения правами, которое имеет владелец, не отмечается в выводимой сводке. Знаком * отмечаются только те права с правом передачи, которые были явно назначены кому-либо.

Примеры

Следующая команда разрешает всем добавлять записи в таблицу films:

```
GRANT INSERT ON films TO PUBLIC;
```

Эта команда даёт пользователю manuel все права для представления kinds:

```
GRANT ALL PRIVILEGES ON kinds TO manuel;
```

Учтите, что если её выполнит суперпользователь или владелец представления kinds, эта команда действительно даст субъекту все права, но если её выполнит обычный пользователь, субъект получит только те права, которые даны этому пользователю с правом передачи.

Включение в роль admins пользователя joe:

```
GRANT admins TO joe;
```

Совместимость

Согласно стандарту SQL, слово PRIVILEGES в указании ALL PRIVILEGES является обязательным. Стандарт SQL не позволяет назначать права сразу для нескольких объектов одной командой.

Postgres Pro позволяет владельцу объекта лишить себя своих обычных прав: например, владелец таблицы может разрешить себе только чтение таблицы, отозвав собственные права INSERT, UPDATE, DELETE и TRUNCATE. В стандарте SQL это невозможно. Это объясняется тем, что Postgres Pro воспринимает права владельца как назначенные им же себе; поэтому их можно и отозвать. В стандарте SQL права владельца даются ему предполагаемой сущностью «_SYSTEM». Так как владелец объекта отличается от «_SYSTEM», лишить себя этих прав он не может.

Согласно стандарту SQL, право с правом передачи можно дать субъекту PUBLIC; однако Postgres Pro может давать право с правом передачи только ролям.

Стандарт SQL разрешает использовать указание GRANTED BY во всех формах GRANT. Postgres Pro поддерживает его только при назначении членства ролей, и при этом только суперпользователи могут использовать его нетривиальным образом.

В стандарте SQL право USAGE распространяется и на другие типы объектов: наборы символов, правила сортировки и преобразования.

В стандарте SQL право USAGE для последовательностей управляет использованием выражения NEXT VALUE FOR, которое равнозначно функции nextval в Postgres Pro. Права SELECT и UPDATE для последовательностей являются расширениями Postgres Pro. То, что право USAGE для последовательностей управляет использованием функции currval, так же относится к расширениям Postgres Pro (как и сама функция).

Права для баз данных, табличных пространств, схем и языков относятся к расширениям Postgres Pro.

См. также

REVOKE, ALTER DEFAULT PRIVILEGES

IMPORT FOREIGN SCHEMA

IMPORT FOREIGN SCHEMA — импортировать определения таблиц со стороннего сервера

Синтаксис

```
IMPORT FOREIGN SCHEMA удалённая_схема
    [ { LIMIT TO | EXCEPT } ( имя_таблицы [, ...] ) ]
    FROM SERVER имя_сервера
    INTO локальная_схема
    [ OPTIONS ( параметр 'значение' [, ... ] ) ]
```

Описание

`IMPORT FOREIGN SCHEMA` создаёт сторонние таблицы, которые представляют таблицы, существующие на стороннем сервере. Новые сторонние таблицы будут принадлежать пользователю, выполняющему команду, и будут содержать корректные определения столбцов и параметры, соответствующие удалённым таблицам.

По умолчанию импортируются все таблицы и представления, существующие в определённой схеме на стороннем сервере. По желанию список таблиц можно ограничить некоторым подмножеством, или исключить из него конкретные таблицы. Новые сторонние таблицы создаются в целевой схеме, которая должна уже существовать.

Чтобы использовать `IMPORT FOREIGN SCHEMA`, необходимо иметь право `USAGE` для стороннего сервера, а также право `CREATE` в целевой схеме.

Параметры

удалённая_схема

Удалённая схема, из которой будут импортированы объекты. Что именно представляет собой удалённая схема, зависит от применяемой обёртки сторонних данных.

`LIMIT TO ( имя_таблицы [, ...] )`

Импортировать только сторонние таблицы с заданными именами. Другие таблицы, существующие в сторонней схеме, будут проигнорированы.

`EXCEPT ( имя_таблицы [, ...] )`

Исключить из импорта указанные сторонние таблицы. Данная команда импортирует все таблицы, существующие в сторонней схеме, за исключением перечисленных в этом предложении.

имя_сервера

Сторонний сервер, с которого импортируется схема.

локальная_схема

Схема, в которой будут созданы импортируемые сторонние таблицы.

`OPTIONS ( параметр 'значение' [, ...] )`

Параметры, которые должны применяться при импорте. Допустимые имена параметров и их значения зависят от обёртки сторонних данных.

Примеры

Импорт определений таблиц из удалённой схемы `foreign_films` на сервере `film_server` с созданием сторонних таблиц в локальной схеме `films`:

```
IMPORT FOREIGN SCHEMA foreign_films
    FROM SERVER film_server INTO films;
```

Та же операция, но импортируются только таблицы `actors` и `directors` (если они существуют):

```
IMPORT FOREIGN SCHEMA foreign_films LIMIT TO (actors, directors)
    FROM SERVER film_server INTO films;
```

Совместимость

Команда `IMPORT FOREIGN SCHEMA` соответствует стандарту SQL, за исключением параметра `OPTIONS`, являющегося расширением Postgres Pro.

См. также

CREATE FOREIGN TABLE, CREATE SERVER

INSERT

INSERT — добавить строки в таблицу

Синтаксис

```
[ WITH [ RECURSIVE ] запрос_WITH [, ...] ]
INSERT INTO имя_таблицы [ AS псевдоним ] [ ( имя_столбца [, ...] ) ]
    [ OVERRIDING { SYSTEM | USER } VALUE ]
    { DEFAULT VALUES | VALUES ( { выражение | DEFAULT } [, ...] ) [, ...] | запрос }
    [ ON CONFLICT [ объект_конфликта ] действие_при_конфликте ]
    [ RETURNING * | выражение_результата [ [ AS ] имя_результата ] [, ...] ]

Здесь допускается объект_конфликта:

    ( { имя_столбца_индекса | ( выражение_индекса ) } [ COLLATE правило_сортировки ]
 [ класс_операторов ] [, ...] ) [ WHERE предикат_индекса ]
    ON CONSTRAINT имя_ограничения

и действие_при_конфликте может быть следующим:

    DO NOTHING
    DO UPDATE SET { имя_столбца = { выражение | DEFAULT } |
                    ( имя_столбца [, ...] ) = [ ROW ] ( { выражение | DEFAULT }
 [, ...] ) |
                    ( имя_столбца [, ...] ) = ( вложенный_SELECT )
                  } [, ...]
              [ WHERE условие ]
```

Описание

`INSERT` добавляет строки в таблицу. Эта команда может добавить одну или несколько строк, сформированных выражениями значений, либо ноль или более строк, выданных дополнительным запросом.

Имена целевых столбцов могут перечисляться в любом порядке. Если список с именами столбцов отсутствует, по умолчанию целевыми столбцами становятся все столбцы заданной таблицы; либо первые *N* из них, если только *N* столбцов поступает от предложения `VALUES` или *запроса*. Значения, получаемые от предложения `VALUES` или *запроса*, связываются с явно или неявно определённым списком столбцов слева направо.

Все столбцы, не представленные в явном или неявном списке столбцов, получат значения по умолчанию, если для них заданы эти значения, либо NULL в противном случае.

Если выражение для любого столбца выдаёт другой тип данных, система попытается автоматически привести его к нужному.

Операция `INSERT` с таблицами без уникальных индексов не блокируется параллельно выполняемыми операциями. В таблицах с уникальными индексами эта операция может блокироваться, если в параллельных сеансах выполняются действия, которые блокируют или изменяют строки, совпадающие с вставляемыми значениями в уникальном индексе; подробнее см. Раздел 56.5. Предложение `ON CONFLICT` позволяет задать действие, заменяющее возникновение ошибки при нарушении ограничения уникальности или ограничения-исключения. (См. описание «Предложение `ON CONFLICT`» ниже.)

С необязательным предложением `RETURNING` команда `INSERT` вычислит и возвратит значения для каждой фактически добавленной строки (или изменённой, если применялось предложение `ON CONFLICT DO UPDATE`). В основном это полезно для получения значений, присвоенных по

умолчанию, например, последовательного номера записи. Однако в этом предложении можно задать любое выражение со столбцами таблицы. Список RETURNING имеет тот же синтаксис, что и список результатов SELECT. В результате будут возвращены те строки, которые были успешно вставлены или изменены. Например, если строка была заблокирована, но не изменена, из-за того, что *условие* в предложении ON CONFLICT DO UPDATE ... WHERE не удовлетворено, эта строка возвращена не будет.

Чтобы добавлять строки в таблицу, необходимо иметь право INSERT для неё. Если присутствует предложение ON CONFLICT DO UPDATE, также требуется иметь право UPDATE для этой таблицы.

Если указывается список столбцов, достаточно иметь право INSERT только для перечисленных столбцов. Аналогично, с предложением ON CONFLICT DO UPDATE достаточно иметь право UPDATE только для столбцов, которые будут изменены. Однако предложение ON CONFLICT DO UPDATE также требует наличия права SELECT для всех столбцов, значения которых считываются в выражениях ON CONFLICT DO UPDATE или в *условии*.

Для применения предложения RETURNING требуется право SELECT для всех столбцов, перечисленных в RETURNING. Если для добавления строк применяется *запрос*, для всех таблиц или столбцов, задействованных в этом запросе, разумеется, необходимо иметь право SELECT.

Параметры

Добавление

В этом разделе рассматриваются параметры, применяемые только при добавлении новых строк. Параметры, применяемые *исключительно* с предложением ON CONFLICT, описываются отдельно.

запрос_WITH

Предложение WITH позволяет задать один или несколько подзапросов, на которые затем можно ссылаться по имени в запросе INSERT. Подробнее об этом см. Раздел 7.8 и SELECT.

Заданный *запрос* (оператор SELECT) также может содержать предложение WITH. В этом случае в *запросе* можно обращаться к обоим *запросам_WITH*, но второй будет иметь приоритет, так как он вложен ближе.

имя_таблицы

Имя существующей таблицы (возможно, дополненное схемой).

псевдоним

Альтернативное имя, заменяющее *имя_таблицы*. Когда указывается этот псевдоним, он полностью скрывает реальное имя таблицы. Это особенно полезно, когда в предложении ON CONFLICT DO UPDATE фигурирует таблица с именем excluded, так как без определения псевдонима это имя будет отдано специальной таблице, представляющей строку, предназначенную для добавления.

имя_столбца

Имя столбца в таблице *имя_таблицы*. Это имя столбца при необходимости может быть дополнено именем вложенного поля или индексом в массиве. (Когда данные вставляются только в некоторые поля столбца составного типа, в другие поля записывается NULL.) Обращаясь к столбцу в предложении ON CONFLICT DO UPDATE, включать имя таблицы в ссылку на целевой столбец не нужно. Например, запись INSERT INTO table_name ... ON CONFLICT DO UPDATE SET table_name.col = 1 некорректна (это согласуется с общим поведением команды UPDATE).

OVERRIDING SYSTEM VALUE

Без этого предложения не допускается задание явного значения (отличного от DEFAULT) для столбца идентификации, определённого с характеристикой GENERATED ALWAYS. Данное предложение перекрывает это ограничение.

`OVERRIDING USER VALUE`

Если указывается это предложение, то значения, представляемые для столбцов идентификации, определённых с характеристикой `GENERATED BY DEFAULT`, игнорируются и вместо них применяются значения, выдаваемые последовательностью по умолчанию.

Это предложение полезно, например, при копировании значений между таблицами. Команда `INSERT INTO tbl2 OVERRIDING USER VALUE SELECT * FROM tbl1` скопирует из `tbl1` все столбцы, кроме столбцов идентификации в `tbl2`, а значения столбцов идентификации в `tbl2` будут сгенерированы последовательностями в `tbl2`.

`DEFAULT VALUES`

Все столбцы получат значения по умолчанию. (Предложение `OVERRIDING` в этой форме не допускается.)

`выражение`

Выражение или значение, которое будет присвоено соответствующему столбцу.

`DEFAULT`

Соответствующий столбец получит значение по умолчанию.

`запрос`

Запрос (оператор `SELECT`), который выдаст строки для добавления в таблицу. Его синтаксис описан в справке оператора SELECT.

`выражение_результата`

Выражение, которое будет вычисляться и возвращаться командой `INSERT` после добавления или изменения каждой строки. В этом выражении можно использовать имена любых столбцов таблицы *`имя_таблицы`*. Чтобы получить все столбцы, достаточно написать `*`.

`имя_результата`

Имя, назначаемое возвращаемому столбцу.

Предложение `ON CONFLICT`

Необязательное предложение `ON CONFLICT` задаёт действие, заменяющее возникновение ошибки при нарушении ограничения уникальности или ограничения-исключения. Для каждой отдельной строки, предложенной для добавления, добавление либо выполняется успешно, либо, если нарушается *решающее* ограничение или индекс, задаваемые как *`объект_конфликта`*, выполняется альтернативное *`действие_конфликта`*. Вариант `ON CONFLICT DO NOTHING` в качестве альтернативного действия просто отменяет добавление строки. Вариант `ON CONFLICT DO UPDATE` изменяет существующую строку, вызвавшую конфликт со строкой, предложенной для добавления.

Задаваемый *`объект_конфликта`* может *выбирать уникальный индекс*. Определение объекта, позволяющее выбрать индекс, включает один или несколько столбцов (их определяет *`имя_столбца_индекса`*) и/или *`выражение_индекса`* и необязательный *`предикат_индекса`*. Все уникальные индексы в таблице *`имя_таблицы`*, которые, без учёта порядка столбцов, содержат в точности столбцы/выражения, определяющие *`объект_конфликта`*, выбираются как решающие индексы. Если указывается *`предикат_индекса`*, он должен, в качестве дополнительного требования выбора, удовлетворять индексам. Заметьте, что это означает, что не частичный уникальный индекс (уникальный индекс без предиката) будет выбран (и будет использоваться в `ON CONFLICT`), если такой индекс удовлетворяет всем остальным критериям. Если попытка выбрать индекс оказывается неудачной, выдаётся ошибка.

`ON CONFLICT DO UPDATE` гарантирует атомарный результат команды `INSERT` или `UPDATE`; при отсутствии внешних ошибок гарантируется один из двух этих исходов, даже при большой параллельной активности. Эта операция также известна как *UPSERT* — «UPDATE или INSERT».

объект_конфликта

Определяет, для какого именно конфликта в ON CONFLICT будет предпринято альтернативное действие, устанавливая *решающие индексы*. Это указание позволяет осуществить *выбор уникального индекса* или явно задаёт имя ограничения. Для ON CONFLICT DO NOTHING *объект_конфликта* может не указываться; в этом случае игнорироваться будут все конфликты с любыми ограничениями (и уникальными индексами). Для ON CONFLICT DO UPDATE *объект_конфликта* *должен* указываться.

действие_при_конфликте

Параметр *действие_при_конфликте* задаёт альтернативное действие в случае конфликта. Это может быть либо DO NOTHING (не делать ничего), либо предложение DO UPDATE (произвести изменение), в котором указываются точные детали операции UPDATE, выполняемой в случае конфликта. Предложения SET и WHERE в ON CONFLICT DO UPDATE могут обращаться к существующей строке по имени таблицы (или псевдониму) или к строке, предлагаемой для добавления, используя специальную таблицу excluded. Для чтения столбцов excluded необходимо иметь право SELECT для соответствующих столбцов в целевой таблице.

Заметьте, что эффект действий всех триггеров уровня строк BEFORE INSERT отражается в значениях excluded, так как в результате этих действий строка может быть исключена из множества добавляемых.

имя_столбца_индекса

Имя столбца в таблице *имя_таблицы*. Используется для выбора решающих индексов. Задаётся в формате CREATE INDEX. Чтобы запрос выполнился, для столбца *имя_столбца_индекса* требуется право SELECT.

выражение_индекса

Подобно указанию *имя_столбца_индекса*, но применяется для выбора индекса по выражениям со столбцами таблицы *имя_таблицы*, фигурирующим в определениях индексов (не по простым столбцам). Задаётся в формате CREATE INDEX. Для всех столбцов, к которым обращается *выражение_индекса*, необходимо иметь право SELECT.

правило_сортировки

Когда задаётся, устанавливает, что соответствующие *имя_столбца_индекса* или *выражение_индекса* должны использовать определённый порядок сортировки, чтобы этот индекс мог быть выбран. Обычно это указание опускается, так как от правил сортировки чаще всего не зависит, произойдёт ли нарушение ограничений или нет. Задаётся в формате CREATE INDEX.

класс_операторов

Когда задаётся, устанавливает, что соответствующие *имя_столбца_индекса* или *выражение_индекса* должны использовать определённый класс, чтобы индекс мог быть выбран. Обычно это указание опускается, потому что семантика *равенства* часто всё равно одна и та же в разных классах операторов типа, или потому что достаточно рассчитывать на то, что заданные уникальные индексы имеют адекватное определение равенства. Задаётся в формате CREATE INDEX.

предикат_индекса

Используется для выбора частичных уникальных индексов. Выбраны могут быть любые индексы, удовлетворяющие предикату (при этом они могут не быть собственно частичными индексами). Задаётся в формате CREATE INDEX. Для всех столбцов, задействованных в *предикате_индекса*, требуется право SELECT.

имя_ограничения

Явно задаёт решающее *ограничение* по имени, что заменяет неявный выбор ограничения или индекса.

условие

Выражение, выдающее значение типа boolean. Изменены будут только те строки, для которых это выражение выдаст true, хотя при выборе действия ON CONFLICT DO UPDATE заблокируются все строки. Заметьте, что *условие* вычисляется в конце, после того как конфликт был признан претендующим на выполнение изменения.

Заметьте, что ограничения-исключения не могут быть решающими в ON CONFLICT DO UPDATE. Во всех случаях в качестве решающих поддерживаются только неоткладываемые (NOT DEFERRABLE) ограничения и уникальные индексы.

Команда INSERT с предложением ON CONFLICT DO UPDATE является «детерминированной». Это означает, что этой команде не разрешено воздействовать на любую существующую строку больше одного раза; в случае такой ситуации возникнет ошибка нарушения мощности множества. Строки, предлагаемые для добавления, не должны дублироваться с точки зрения атрибутов, ограничиваемых решающим индексом или ограничением.

> **Подсказка**
>
> Часто предпочтительнее использовать неявный выбор уникального индекса вместо непосредственного указания ограничения в виде ON CONFLICT ON CONSTRAINT *имя_ограничения*. Выбор продолжит корректно работать, когда нижележащий индекс будет заменён другим более или менее равнозначным индексом методом наложения, например, с использованием CREATE UNIQUE INDEX ... CONCURRENTLY и последующим удалением заменяемого индекса.

Выводимая информация

В случае успешного завершения INSERT возвращает метку команды в виде

INSERT *oid число*

Здесь *число* представляет количество добавленных или изменённых строк. Если *число* равняется одному, а целевая таблица содержит oid, то в качестве *oid* выводится OID, назначенный добавленной строке. Эта одна строка должна быть добавлена, но не изменена. В противном случае в качестве *oid* выводится ноль.

Если команда INSERT содержит предложение RETURNING, её результат будет похож на результат оператора SELECT (с теми же столбцами и значениями, что содержатся в списке RETURNING), полученный для строк, добавленных или изменённых этой командой.

Замечания

Если целевая таблица является секционированной, каждая строка перенаправляется в соответствующую секцию и вставляется в неё. Если целевая таблица является секцией и какая-либо из входных строк нарушает ограничение этой секции, происходит ошибка.

Примеры

Добавление одной строки в таблицу films:

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
```

В этом примере столбец len опускается и, таким образом, получает значение по умолчанию:

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

В этом примере для столбца с датой задаётся указание DEFAULT, а не явное значение:

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes');
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama');
```

Добавление строки, полностью состоящей из значений по умолчанию:

```
INSERT INTO films DEFAULT VALUES;
```

Добавление нескольких строк с использованием многострочного синтаксиса `VALUES`:

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

В этом примере в таблицу `films` вставляются некоторые строки из таблицы `tmp_films`, имеющей ту же структуру столбцов, что и `films`:

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

Этот пример демонстрирует добавление данных в столбцы с типом массива:

```
-- Создание пустого поля 3x3 для игры в крестики-нолики
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1, '{{" "," "," "},{" "," "," "},{" "," "," "}}');
-- Указания индексов в предыдущей команда могут быть опущены
INSERT INTO tictactoe (game, board)
    VALUES (2, '{{X," "," "},{" ",O," "},{" ",X," "}}');
```

Добавление одной строки в таблицу `distributors` и получение последовательного номера, сгенерированного благодаря указанию `DEFAULT`:

```
INSERT INTO distributors (did, dname) VALUES (DEFAULT, 'XYZ Widgets')
   RETURNING did;
```

Увеличение счётчика продаж для продавца, занимающегося компанией Acme Corporation, и сохранение всей изменённой строки вместе с текущим временем в таблице журнала:

```
WITH upd AS (
  UPDATE employees SET sales_count = sales_count + 1 WHERE id =
    (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation')
    RETURNING *
)
INSERT INTO employees_log SELECT *, current_timestamp FROM upd;
```

Добавить дистрибьюторов или изменить существующие данные должным образом. Предполагается, что в таблице определён уникальный индекс, ограничивающий значения в столбце `did`. Заметьте, что для обращения к значениям, изначально предлагаемым для добавления, используется специальная таблица `excluded`:

```
INSERT INTO distributors (did, dname)
    VALUES (5, 'Gizmo Transglobal'), (6, 'Associated Computing, Inc')
    ON CONFLICT (did) DO UPDATE SET dname = EXCLUDED.dname;
```

Добавить дистрибьютора или не делать ничего для строк, предложенных для добавления, если уже есть существующая исключающая строка (строка, содержащая конфликтующие значения в столбце или столбцах после срабатывания триггеров перед добавлением строки). В данном примере предполагается, что определён уникальный индекс, ограничивающий значения в столбце `did`:

```
INSERT INTO distributors (did, dname) VALUES (7, 'Redline GmbH')
    ON CONFLICT (did) DO NOTHING;
```

Добавить дистрибьюторов или изменить существующие данные должным образом. В данном примере предполагается, что в таблице определён уникальный индекс, ограничивающий значения в столбце `did`. Предложение `WHERE` позволяет ограничить набор фактически изменяемых строк (однако любая существующая строка, не подлежащая изменению, всё же будет заблокирована):

```
-- Не менять данные существующих дистрибьюторов в зависимости от почтового индекса
INSERT INTO distributors AS d (did, dname) VALUES (8, 'Anvil Distribution')
    ON CONFLICT (did) DO UPDATE
    SET dname = EXCLUDED.dname || ' (formerly ' || d.dname || ')'
    WHERE d.zipcode <> '21201';

-- Указать имя ограничения непосредственно в операторе (связанный индекс
-- применяется для принятия решения о выполнении действия DO NOTHING)
INSERT INTO distributors (did, dname) VALUES (9, 'Antwerp Design')
    ON CONFLICT ON CONSTRAINT distributors_pkey DO NOTHING;
```

Добавить дистрибьютора, если возможно; в противном случае не делать ничего (`DO NOTHING`). В данном примере предполагается, что в таблице определён уникальный индекс, ограничивающий значения в столбце `did` по подмножеству строк, в котором логический столбец `is_active` содержит `true`:

```
-- Этот оператор может выбрать частичный уникальный индекс по "did"
-- с предикатом "WHERE is_active", а может и просто использовать
-- обычное ограничение уникальности по столбцу "did"
INSERT INTO distributors (did, dname) VALUES (10, 'Conrad International')
    ON CONFLICT (did) WHERE is_active DO NOTHING;
```

Совместимость

`INSERT` соответствует стандарту SQL, но предложение `RETURNING` относится к расширениям Postgres Pro, как и возможность применять `WITH` с `INSERT` и возможность задавать альтернативное действие с `ON CONFLICT`. Кроме того, ситуация, когда список столбцов опущен, но не все столбцы получают значения из предложения `VALUES` или *`запроса`*, стандартом не допускается.

В стандарте SQL говорится, что предложение `OVERRIDING SYSTEM VALUE` может присутствовать, только если существует столбец идентификации, для которого всегда генерируется значение. Postgres Pro допускает это предложение в любом случае и игнорирует его в случае неприменимости.

Возможные ограничения предложения *`запрос`* описаны в справке SELECT.

LISTEN

LISTEN — ожидать уведомления

Синтаксис

```
LISTEN канал
```

Описание

`LISTEN` регистрирует текущий сеанс для получения уведомлений через канал с заданным именем (*канал*). Если текущий сеанс уже зарегистрирован и ожидает уведомлений через этот канал, ничего не происходит.

Когда вызывается команда `NOTIFY` *канал* (в текущем или другом сеансе, подключённом к той же базе данных), все сеансы, ожидающие уведомления через заданный канал, получают уведомление и каждый, в свою очередь, передаёт его подключённому клиентскому приложению.

Сеанс может отказаться от получения уведомлений через определённый канал с помощью команды `UNLISTEN`. Кроме того, подписка на любые уведомления автоматически отменяется при завершении сеанса.

Способ получения уведомлений клиентским приложением определяется программным интерфейсом Postgres Pro, который оно использует. Приложение, использующее библиотеку libpq, выполняет команду `LISTEN` как обычную команду SQL, а затем оно должно периодически вызывать функцию `PQnotifies`, чтобы проверить, не поступили ли новые уведомления. Другие интерфейсы, например libpgtcl, предоставляют более высокоуровневые методы для обработки событий уведомлений; на самом деле с libpgtcl разработчик приложения даже не должен непосредственно выполнять команды `LISTEN` и `UNLISTEN`. За дополнительными подробностями обратитесь к документации интерфейса, который вы используете.

В описании NOTIFY использование `LISTEN` и `NOTIFY` рассматривается более подробно.

Параметры

канал

Имя канала уведомлений (любой идентификатор).

Замечания

`LISTEN` начинает действовать при фиксировании транзакции. Если `LISTEN` или `UNLISTEN` выполняется в транзакции, которая затем откатывается, состояние подписки этого сеанса на уведомления не меняется.

Транзакция, в которой выполняется `LISTEN`, не может быть подготовлена для двухфазной фиксации.

Примеры

Демонстрация процедуры ожидания/получения уведомления в psql:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

Совместимость

Оператор `LISTEN` отсутствует в стандарте SQL.

См. также

NOTIFY, UNLISTEN

LOAD

LOAD — загрузить файл разделяемой библиотеки

Синтаксис

```
LOAD 'имя_файла'
```

Описание

Эта команда загружает файл разделяемой библиотеки в адресное пространство сервера Postgres Pro. Если указанный файл был загружен ранее, эта команда не делает ничего. Файлы библиотек, содержащие функции на C, загружаются автоматически при первом вызове любой из этих функций. Поэтому явно выполнять `LOAD` обычно требуется только для загрузки библиотек, которые изменяют поведение сервера, внедряя свои обработчики, а не предоставляют некоторый набор функций.

Имя файла библиотеки обычно задаётся собственно именем файла, а полный путь определяется при просмотре пути поиска библиотек сервера (задаваемого в dynamic_library_path). Также в качестве имени может быть передан непосредственно полный путь. В любом случае расширение имени файла, стандартное для файлов разделяемых библиотек на данной платформе, можно опустить. Дополнительную информацию по этой теме можно найти в Подразделе 35.9.1.

Обычные пользователи (не суперпользователи) могут использовать `LOAD` только для загрузки файлов библиотек, расположенных в `$libdir/plugins/` — заданное *имя_файла* должно начинаться именно с этой строки. (Ответственность за то, чтобы в этом каталоге находились только «безопасные» библиотеки, лежит на администраторе баз данных.)

Совместимость

`LOAD` является расширением Postgres Pro.

См. также

CREATE FUNCTION

LOCK

LOCK — заблокировать таблицу

Синтаксис

```
LOCK [ TABLE ] [ ONLY ] имя [ * ] [, ...] [ IN режим_блокировки MODE ] [ NOWAIT ]

Где режим_блокировки может быть следующим:

    ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
    | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

Описание

`LOCK TABLE` получает блокировку на уровне таблицы, при необходимости ожидая освобождения таблицы от других конфликтующих блокировок. Если указано `NOWAIT`, `LOCK TABLE` не ждёт, пока таблица освободится: если блокировку нельзя получить немедленно, команда прерывается и выдаётся ошибка. Как только блокировка получена, она удерживается до завершения текущей транзакции. (Команды `UNLOCK TABLE` не существует; блокировки всегда освобождаются в конце транзакции.)

Запрашивая автоматические блокировки для команд, работающих с таблицами, Postgres Pro всегда выбирает наименее ограничивающий режим блокировки из возможных. Оператор `LOCK TABLE` предназначен для случаев, когда требуется более сильная блокировка. Например, предположим, что приложение выполняет транзакцию на уровне изоляции `READ COMMITTED` и оно должно получать неизменные данные на протяжении всей транзакции. Для достижения этой цели можно получить для таблицы блокировку в режиме `SHARE`, прежде чем обращаться к ней. В результате параллельные изменения данных будут исключены и при последующих чтениях будет получено стабильное представление зафиксированных данных, так как режим блокировки `SHARE` конфликтует с блокировкой `ROW EXCLUSIVE`, запрашиваемой при записи, а `LOCK TABLE имя IN SHARE MODE` будет ждать, пока параллельные транзакции с блокировкой `ROW EXCLUSIVE` не будут зафиксированы или отменены. Таким образом, в момент получения такой блокировки не останется ни одной открытой незафиксированной операции записи; кроме того, никто не сможет записывать в таблицу, пока блокировка не будет снята.

Чтобы получить похожий эффект в транзакции на уровне изоляции `REPEATABLE READ` или `SERIALIZABLE`, необходимо выполнить оператор `LOCK TABLE` перед первым `SELECT` или оператором, изменяющим данные. Представление данных для транзакции уровня `REPEATABLE READ` или `SERIALIZABLE` будет заморожено в момент, когда начнёт выполняться этот запрос. Если команда `LOCK TABLE` выполняется в транзакции позже, она так же исключает параллельную запись, но не даёт гарантии, что транзакция будет читать последние зафиксированные данные.

Если в транзакции такого рода требуется изменять данные в таблице, для неё следует использовать режим блокировки `SHARE ROW EXCLUSIVE` вместо `SHARE`. Этот режим гарантирует, что в один момент времени будет выполняться только одна транзакция такого типа. Без этого ограничения возможна взаимоблокировка: две транзакции могут одновременно получить блокировки `SHARE`, после чего они не смогут получить блокировку `ROW EXCLUSIVE`, чтобы собственно выполнить изменения. (Заметьте, что собственные блокировки транзакции никогда не конфликтуют, так что транзакция может получить блокировку `ROW EXCLUSIVE`, когда она владеет блокировкой `SHARE` — но не тогда, когда блокировку `SHARE` удерживает другая транзакция.) Чтобы не допустить взаимоблокировок, убедитесь, что все транзакции запрашивают блокировки одних объектов в одинаковом порядке, и если для одного объекта запрашиваются блокировки в разных режимах, транзакции всегда должны запрашивать самую строгую блокировку.

Дополнительно о режимах и стратегиях блокировки можно узнать в Разделе 13.3.

Параметры

имя

Имя (возможно, дополненное схемой) существующей таблицы, для которой запрашивается блокировка. Если перед именем таблицы указано ONLY, блокируется только заданная таблица. Без ONLY блокируется и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно также добавить необязательное указание *, чтобы явно обозначить, что блокировка затрагивает и все дочерние таблицы.

Команда LOCK TABLE a, b; равнозначна последовательности LOCK TABLE a; LOCK TABLE b;. Таблицы блокируются по одной в порядке, заданном в команде LOCK TABLE.

режим_блокировки

Режим блокировки определяет, с какой блокировкой будет конфликтовать данная. Режимы блокировок описаны в Разделе 13.3.

Если режим блокировки не указан, применяется самый строгий режим, ACCESS EXCLUSIVE.

NOWAIT

Указывает, что LOCK TABLE не должна ожидать освобождения конфликтующих блокировок: если запрошенная блокировка не может быть получена немедленно, транзакция прерывается.

Замечания

LOCK TABLE ... IN ACCESS SHARE MODE требует наличия права SELECT в целевой таблице. LOCK TABLE ... IN ROW EXCLUSIVE MODE требует наличия прав INSERT, UPDATE, DELETE или TRUNCATE для целевой таблицы. Все другие формы LOCK требуют наличия права UPDATE, DELETE или TRUNCATE на уровне таблицы.

Вне блока транзакции команда LOCK TABLE бесполезна: блокировка сохранится только до завершения операции. Поэтому Postgres Pro выдаёт ошибку при попытке применить LOCK не в блоке транзакции. Чтобы определить блок транзакции, используйте BEGIN и COMMIT (или ROLLBACK).

LOCK TABLE может устанавливать только блокировки на уровне таблицы, так что все имена режимов, включающие слово ROW (строка), не совсем корректны. Следует воспринимать их так, что в этих режимах пользователь намеревается получать в заблокированной таблице блокировки уровня строк. Также учтите, что в режиме ROW EXCLUSIVE устанавливается разделяемая блокировка таблицы. Заметьте, что применительно к LOCK TABLE все режимы блокировки действуют одинаково, отличаются только правила, определяющие, какой режим с каким конфликтует. Чтобы узнать, как получить блокировку именно на уровне строк, обратитесь к Подразделу 13.3.2 и Подразделу «Предложение блокировки» в справочной документации SELECT.

Примеры

Получение блокировки SHARE для первичного ключа таблицы при добавлении записи в подчинённую таблицу:

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- Если запись не будет возвращена, произойдёт откат транзакции
INSERT INTO films_user_comments VALUES
    (_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

Установление блокировки SHARE ROW EXCLUSIVE в таблице первичного ключа перед выполнением операции удаления:

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
    (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

Совместимость

Команда `LOCK TABLE` отсутствует в стандарте SQL, в нём уровни изоляции транзакции определяются командой `SET TRANSACTION`. Postgres Pro поддерживает и этот вариант; подробнее это описано в SET TRANSACTION.

За исключением `ACCESS SHARE`, `ACCESS EXCLUSIVE` и `SHARE UPDATE EXCLUSIVE`, режимы блокировки в Postgres Pro и синтаксис `LOCK TABLE` совместимы с теми, что представлены в СУБД Oracle.

MOVE

MOVE — переместить курсор

Синтаксис

```
MOVE [ направление ] [ FROM | IN ] имя_курсора

Здесь направление может принимать следующие значения:

    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE число
    RELATIVE число
    число
    ALL
    FORWARD
    FORWARD число
    FORWARD ALL
    BACKWARD
    BACKWARD число
    BACKWARD ALL
```

Описание

`MOVE` перемещает курсор, не получая данные. Команда `MOVE` работает точно так же, как `FETCH`, но она не возвращает данные строк, а только перемещает курсор.

Команда `MOVE` поддерживает те же параметры, что и `FETCH`; за подробным описанием её синтаксиса и использования обратитесь к FETCH.

Выводимая информация

В случае успешного завершения, `MOVE` возвращает метку команды в виде

```
MOVE число
```

Здесь *число* показывает количество строк, которое бы выдала команда `FETCH` с такими же параметрами (оно может быть нулевым).

Примеры

```
BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;

-- Пропустить первые 5 строк:
MOVE FORWARD 5 IN liahona;
MOVE 5

-- Выбрать 6-ю строку из курсора liahona:
FETCH 1 FROM liahona;
 code  | title  | did | date_prod  |  kind  |  len
-------+--------+-----+------------+--------+-------
 P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)
```

```
-- Закрыть курсор liahona и завершить транзакцию:
CLOSE liahona;
COMMIT WORK;
```

Совместимость

Оператор `MOVE` отсутствует в стандарте SQL.

См. также

CLOSE, DECLARE, FETCH

NOTIFY

NOTIFY — сгенерировать уведомление

Синтаксис

```
NOTIFY канал [ , сообщение ]
```

Описание

Команда NOTIFY отправляет событие уведомления вместе с дополнительной строкой «сообщения» всем клиентским приложениям, которые до этого выполнили в текущей базе данных LISTEN *канал* с указанным именем канала. Уведомления видны всем пользователям.

NOTIFY предоставляет простой механизм межпроцессного взаимодействия для множества процессов, работающих с одной базой данных Postgres Pro. Вместе с уведомлением может быть передана строка сообщения, а передавая дополнительные данные через таблицы базы данных, можно создать более высокоуровневые механизмы обмена структурированными данными.

Информация, передаваемая клиенту с уведомлением, включает имя канала уведомлений, PID серверного процесса, управляющего сеансом, который выдал уведомление, и строку сообщения (она будет пустой, если сообщение не задано).

Выбор подходящих имён каналов и их назначения — дело проектировщика базы данных. Обычно имя канала совпадает с именем какой-либо таблицы в базе, а событие уведомления по сути означает «я изменила эту таблицу, посмотрите, что она содержит теперь». Однако команды NOTIFY и LISTEN не навязывают именно такой подход. Например, проектировщик базы данных может выбрать разные имена каналов, чтобы сигнализировать о разных типах изменений в одной таблице. Кроме того, строку сообщения тоже можно использовать для выделения различных событий.

Если требуется сигнализировать о факте изменений в определённой таблице, используя NOTIFY, можно применить полезный программный приём — поместить NOTIFY в триггер уровня оператора, который будет срабатывать при изменениях в таблице. При таком подходе уведомление будет выдаваться автоматически, так что прикладной программист не рискует случайно оставить какое-либо изменение без уведомления.

Транзакции оказывают значительное влияние на работу NOTIFY. Во-первых, если NOTIFY выполняется внутри транзакции, уведомления доставляются получателям после фиксирования транзакции и только в этом случае. Это разумно, так как в случае прерывания транзакции действие всех команд в ней аннулируется, включая NOTIFY. Однако это может обескуражить тех, кто ожидает, что уведомления будут приходить немедленно. Во-вторых, если ожидающий сеанс получает уведомление внутри транзакции, это событие не будет доставлено подключённому клиенту до завершения (фиксации или отката) транзакции. Это опять же объясняется тем, что если уведомление будет доставлено в рамках транзакции, которая затем будет прервана, может возникнуть желание как-то отменить его — но сервер не может «забрать назад» уведомление после того, как оно было отправлено клиенту. Поэтому уведомления доставляются только между транзакциями. Учитывая вышесказанное, в приложениях, применяющих NOTIFY для сигнализации в реальном времени, следует минимизировать размер транзакций.

Если в рамках одной транзакции в один канал поступило несколько уведомлений с одинаковым сообщением, сервер может решить доставить только одно уведомление. Если же сообщения различаются, уведомления будут всегда доставлены по отдельности. Так же уведомления, поступающие от разных транзакций, никогда не будут объединены в одно. Не считая фильтрации последующих экземпляров дублирующихся уведомлений, NOTIFY гарантирует, что уведомления от одной транзакции всегда поступают в том же порядке, в каком были отправлены. Также гарантируется, что сообщения от разных транзакций поступают в порядке фиксации этих транзакций.

Часто бывает, что клиент, выполнивший NOTIFY, ожидает уведомления на этом же канале. В этом случае он получит своё же уведомление, как и любой другой сеанс, ожидающий уведомления. В зависимости от логики приложения, это может привести к бессмысленным операциям, например, поиску изменений в таблице, которые и были внесены этим же сеансом. Этой дополнительной работы можно избежать, если проверить, не совпадает ли PID сигнализирующего процесса (указанный в данных события) с собственным PID сеанса (его можно узнать, обратившись к libpq). Если они совпадают, значит сеанс получил уведомление о собственных действиях, так что его можно игнорировать.

Параметры

канал

Имя канала для передачи уведомления (любой идентификатор).

сообщение

Строка «сообщения», которая будет передана вместе с уведомлением. Она должна задаваться простой текстовой константой. В стандартной конфигурации её длина должна быть меньше 8000 байт. (Если требуется передать двоичные данные или большой объём информации, лучше поместить их в таблицу базы данных и передать ключ этой записи.)

Замечания

Уведомления, которые были отправлены, но ещё не обработаны всеми ожидающими сеансами, содержатся в очереди. Если эта очередь переполняется, транзакции, в которых вызывается NOTIFY, будут завершены ошибкой при попытке фиксации. Очередь довольно велика (8 ГБ в стандартной конфигурации), так что её размера должно хватать практически во всех случаях, но если в сеансе выполняется LISTEN, а затем продолжается очень длительная транзакция, очередь не очищается. Как только эта очередь заполняется наполовину, в журнал записываются предупреждения, в которых указывается, какой сеанс препятствует очистке очереди. В этом случае следует добиться завершения текущей транзакции в указанном сеансе, чтобы очередь была очищена.

Функция pg_notification_queue_usage показывает, какой процент очереди в данный момент занят ожидающими уведомлениями. За дополнительными сведениями обратитесь к Разделу 9.25.

Транзакция, в которой выполняется NOTIFY, не может быть подготовлена для двухфазной фиксации.

pg_notify

Также отправить уведомление можно, используя функцию pg_notify(text, text). Эта функция принимает в первом аргументе имя канала, а во втором текст сообщения. Гораздо удобнее использовать её, когда требуется работать с динамическими именами каналов и сообщениями.

Примеры

Демонстрация процедуры ожидания/получения уведомления в psql:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
NOTIFY virtual, 'This is the payload';
Asynchronous notification "virtual" with payload "This is the payload" received from
 server process with PID 8448.

LISTEN foo;
SELECT pg_notify('fo' || 'o', 'pay' || 'load');
Asynchronous notification "foo" with payload "payload" received from server process
 with PID 14728.
```

Совместимость

Оператор `NOTIFY` отсутствует в стандарте SQL.

См. также

LISTEN, UNLISTEN

PREPARE

PREPARE — подготовить оператор к выполнению

Синтаксис

```
PREPARE имя [ ( тип_данных [, ...] ) ] AS оператор
```

Описание

`PREPARE` создаёт подготовленный оператор. Подготовленный оператор представляет собой объект на стороне сервера, позволяющий оптимизировать производительность приложений. Когда выполняется `PREPARE`, указанный оператор разбирается, анализируется и переписывается. При последующем выполнении команды `EXECUTE` подготовленный оператор планируется и исполняется. Такое разделение труда исключает повторный разбор запроса, при этом позволяет выбрать наилучший план выполнения в зависимости от определённых значений параметров.

Подготовленные операторы могут принимать параметры — значения, которые подставляются в оператор, когда он собственно выполняется. При создании подготовленного оператора к этим параметрам можно обращаться по порядковому номеру, используя запись `$1`, `$2` и т. д. Дополнительно можно указать список соответствующих типов данных параметров. Если тип данных параметра не указан или объявлен как `unknown` (неизвестный), тип выводится из контекста, в котором этот параметр используется впервые (если это возможно). При выполнении оператора фактические значения параметров передаются команде `EXECUTE`. За подробностями обратитесь к EXECUTE.

Подготовленные операторы существуют только в рамках текущего сеанса работы с БД. Когда сеанс завершается, система забывает подготовленный оператор, так что его надо будет создать снова, чтобы использовать дальше. Это также означает, что один подготовленный оператор не может использоваться одновременно несколькими клиентами базы данных; но каждый клиент может создать собственный подготовленный оператор и использовать его. Освободить подготовленный оператор можно вручную, выполнив команду DEALLOCATE.

Подготовленные операторы потенциально дают наибольший выигрыш в производительности, когда в одном сеансе выполняется большое число однотипных операторов. Отличие в производительности особенно значительно, если операторы достаточно сложны для планирования или перезаписи, например, когда в запросе объединяется множество таблиц или необходимо применить несколько правил. Если оператор относительно прост в этом плане, но сложен для выполнения, выигрыш от использования подготовленных операторов будет менее заметным.

Параметры

имя

Произвольное имя, назначаемое данному подготовленному оператору. Оно должно быть уникальным в рамках одного сеанса; это имя затем используется для выполнения или освобождения ранее подготовленного оператора.

тип_данных

Тип данных параметра подготовленного оператора. Если тип данных конкретного параметра не задан или задан как `unknown`, он будет выводиться из контекста, в котором этот параметр используется впервые. Для обращения к параметрам в самом подготовленном операторе используется запись `$1`, `$2` и т. д.

оператор

Любой оператор `SELECT`, `INSERT`, `UPDATE`, `DELETE` или `VALUES`.

Замечания

Подготовленные операторы могут использовать общие планы, а не перестраивать план для каждого набора переданных значений EXECUTE. Для подготовленных операторов без параметров это происходит сразу; иначе общий план выбирается после пяти и более выполнений, при которых получаются планы с ожидаемой средней стоимостью (включая издержки планирования), превышающей оценку стоимости общего плана. Когда общий план выбран, он будет использоваться до конца жизни подготовленного оператора. При использовании значений EXECUTE, которые редко встречаются в столбцах со множеством дублирующихся значений, могут быть построены специализированные планы настолько выгоднее общего плана, что даже с издержками планирования общий план может не использоваться никогда.

Для общего плана предполагается, что значения, передаваемые в EXECUTE, являются уникальными значениями в столбце и что эти значения распределены равномерно. Например, если в статистике записаны три различных значения столбца, с общим планом предполагается, что проверке на равенство для столбца будут соответствовать 33% обработанных строк. Статистика по столбцам также позволяет общим планам точно вычислять избирательность для уникальных столбцов. Сравнения по столбцам с неоднородным распределением и указания несуществующих значений влияют на среднюю стоимость плана и следовательно, на то, будет ли выбран общий план и когда.

Чтобы узнать, какой план выполнения выбирает Postgres Pro для подготовленного оператора, воспользуйтесь EXPLAIN (например, напишите EXPLAIN EXECUTE). Если применяется общий план, он будет содержать символы параметров $*n*, тогда как в специализированном плане будут подставлены фактические значения параметров. Оценки строк в общем плане отражают избирательность, вычисленную для конкретных параметров.

Более подробно о планировании запросов и статистике, которую собирает Postgres Pro для этих целей, можно узнать в документации ANALYZE.

Хотя основной смысл подготовленных операторов в том, чтобы избежать многократного разбора и планирования оператора, Postgres Pro будет принудительно заново анализировать и планировать выполнение оператора всякий раз, когда объекты базы данных, задействованные в операторе, подвергаются изменениям определения (DDL) со времени предыдущего использования подготовленного оператора. Кроме того, если от одного использования оператора к другому меняется значение search_path, оператор будет так же разобран заново с новым search_path. (Последнее поведение появилось в PostgreSQL 9.3.) С этими правилами использование подготовленного оператора по сути почти не отличается от выполнения одного и того же запроса снова и снова, но даёт выигрыш по скорости (если определения объектов не меняются), особенно если оптимальный план от раза к разу не меняется. Однако различия всё же могут проявиться — например, когда оператор обращается к таблице по неполному имени, а затем в схеме, стоящей в пути search_path раньше, создаётся другая таблица с таким же именем, автоматический пересмотр запроса не происходит, так как никакой объект в определении оператора не изменился. Однако если автоматический пересмотр произойдёт в результате других изменений, при последующем выполнении запроса будет задействована новая таблица.

Получить список всех доступных в сеансе подготовленных операторов можно, обратившись к системному представлению pg_prepared_statements.

Примеры

Создание подготовленного оператора для команды INSERT, который затем выполняется:

```
PREPARE fooplan (int, text, bool, numeric) AS
    INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

Создание подготовленного оператора для команды SELECT, который затем выполняется:

```
PREPARE usrrptplan (int) AS
    SELECT * FROM users u, logs l WHERE u.usrid=$1 AND u.usrid=l.usrid
```

```
    AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

Заметьте, что тип данных второго параметра не указывается, так что он выводится из контекста, в котором используется `$2`.

Совместимость

В стандарте SQL есть оператор `PREPARE`, но он предназначен только для применения во встраиваемом SQL. Эта версия оператора `PREPARE` имеет также несколько другой синтаксис.

См. также

DEALLOCATE, EXECUTE

PREPARE TRANSACTION

PREPARE TRANSACTION — подготовить текущую транзакцию для двухфазной фиксации

Синтаксис

```
PREPARE TRANSACTION id_транзакции
```

Описание

`PREPARE TRANSACTION` подготавливает текущую транзакцию для двухфазной фиксации. После этой команды транзакция перестаёт быть связанной с текущим сеансом; её состояние полностью сохраняется на диске, и есть очень большая вероятность, что она будет успешно зафиксирована, даже если до этого времени работа базы данных будет прервана аварийно.

Подготовленную транзакцию затем можно зафиксировать или отменить командами COMMIT PREPARED и ROLLBACK PREPARED, соответственно. Эти команды можно вызывать из любого сеанса, не только из того, в котором эта транзакция создавалась.

С точки зрения сеанса, выполняющего команду, `PREPARE TRANSACTION` не отличается от `ROLLBACK`: после её выполнения не активна никакая транзакция, а результат действия подготовленной транзакции становится невидимым (Он окажется видимым снова, если транзакция будет зафиксирована.)

Если при выполнении команды `PREPARE TRANSACTION` по какой-то причине происходит сбой, команда действует как `ROLLBACK`: текущая транзакция откатывается.

Параметры

id_транзакции

Произвольный идентификатор, по которому затем на эту транзакцию будут ссылаться команды `COMMIT PREPARED` или `ROLLBACK PREPARED`. Идентификатор должен задаваться строковой константой не длиннее 200 байт и должен отличаться от идентификаторов любых других подготовленных на данный момент транзакций.

Замечания

`PREPARE TRANSACTION` не предназначена для использования в приложениях или интерактивных сеансах. Её задача — дать возможность внешнему менеджеру транзакций выполнять атомарные глобальные транзакции, охватывающие несколько баз данных или другие транзакционные ресурсы. Обычно применять `PREPARE TRANSACTION` следует только при разработке собственного менеджера транзакций.

Эта команда должна выполняться внутри блока транзакции. Начинает блок транзакции команда BEGIN.

В настоящее время команда `PREPARE` неспособна подготавливать транзакции, в которых выполнялись какие-либо действия с временными таблицами или во временном пространстве имён сеанса, создавались курсоры `WITH HOLD` либо выполнялись команды `LISTEN`, `UNLISTEN` или `NOTIFY`. Эти функции слишком тесно связаны с текущим сеансом, так что в подготовленной транзакции они не были бы полезны.

Если транзакция меняет какие-либо параметры времени выполнения командой `SET` (без указания `LOCAL`), их значения сохраняются после `PREPARE TRANSACTION` и не зависят от последующих команд `COMMIT PREPARED` и `ROLLBACK PREPARED`. Так что в этом отношении `PREPARE TRANSACTION` больше похожа на `COMMIT`, чем на `ROLLBACK`.

Все существующие в текущий момент подготовленные транзакции показываются в системном представлении `pg_prepared_xacts`.

> **Внимание**
>
> Оставлять транзакции в подготовленном состоянии на долгое время не рекомендуется. Это повлияет на способность команды VACUUM высвобождать пространство, а в крайнем случае может привести к отключению базы данных для предотвращения зацикливания ID транзакций (см. Подраздел 23.1.5). Также учтите, что транзакция продолжит удерживать все свои блокировки. Это сделано с расчётом на то, что подготовленная транзакция будет зафиксирована или отменена как только внешний менеджер транзакций убедится, что все другие базы данных так же готовы к фиксации.
>
> В отсутствие настроенного внешнего менеджера транзакций, который бы отслеживал подготовленные транзакции и своевременно закрывал их, лучше вовсе отключить поддержку подготовленных транзакций, установив max_prepared_transactions равным нулю. Это не позволит случайно создать подготовленные транзакции, которые могут быть забыты и в конце концов станут причиной проблем.

Примеры

Текущая транзакция подготавливается для двухфазной фиксации, при этом ей назначается идентификатор foobar:

```
PREPARE TRANSACTION 'foobar';
```

Совместимость

Оператор PREPARE TRANSACTION является расширением Postgres Pro. Он предназначен для использования внешними системами управления транзакциями, некоторые из которых работают по стандартам (например, X/Open XA), но сторона SQL в этих системах не стандартизирована.

См. также

COMMIT PREPARED, ROLLBACK PREPARED

REASSIGN OWNED

REASSIGN OWNED — сменить владельца объектов базы данных, принадлежащих заданной роли

Синтаксис

```
REASSIGN OWNED BY { старая_роль | CURRENT_USER | SESSION_USER } [, ...]
               TO { новая_роль | CURRENT_USER | SESSION_USER }
```

Описание

`REASSIGN OWNED` указывает системе сменить владельца объектов баз данных, принадлежащих одной из *старых_ролей*, на *новую_роль*.

Параметры

старая_роль

Имя роли. Все объекты в текущей базе данных и все общие объекты (базы данных, табличные пространства), принадлежащие этой роли, станут принадлежать *новой_роли*.

новая_роль

Имя роли, которая станет новым владельцем затронутых объектов.

Замечания

`REASSIGN OWNED` часто применяется при подготовке к удалению одной или нескольких ролей. Так как команда `REASSIGN OWNED` затрагивает объекты только в текущей базе данных, обычно её нужно выполнять в каждой базе данных, которая содержит объекты, принадлежащие удаляемой роли.

Для выполнения `REASSIGN OWNED` требуются права и для исходной, и для целевой роли.

Команда DROP OWNED предлагает альтернативное решение, просто удаляя все объекты базы данных, принадлежащие одной или нескольким заданным ролям.

Команда `REASSIGN OWNED` не затрагивает никакие права, которые даны *старым_ролям* для объектов, им не принадлежащим. Также она не затрагивает права по умолчанию, установленные командой `ALTER DEFAULT PRIVILEGES`. Отозвать эти права можно, воспользовавшись командой `DROP OWNED`.

За подробностями обратитесь к Разделу 20.4.

Совместимость

Оператор `REASSIGN OWNED` является расширением Postgres Pro.

См. также

DROP OWNED, DROP ROLE, ALTER DATABASE

REFRESH MATERIALIZED VIEW

REFRESH MATERIALIZED VIEW — заменить содержимое материализованного представления

Синтаксис

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] имя
    [ WITH [ NO ] DATA ]
```

Описание

`REFRESH MATERIALIZED VIEW` полностью заменяет содержимое материализованного представления. Эту команду разрешено выполнять только владельцам мат. представления. Старое его содержимое при этом аннулируется. Если добавлено указание `WITH DATA` (или нет никакого), нижележащий запрос выполняется и выдаёт новые данные, так что материализованное представление остаётся в сканируемом состоянии. Если указано `WITH NO DATA`, новые данные не выдаются, и оно оказывается в несканируемом состоянии.

Указать `CONCURRENTLY` вместе с `WITH NO DATA` нельзя.

Параметры

`CONCURRENTLY`

Обновить материализованное представление, не блокируя параллельные выборки из него. Без данного параметра обновление, затрагивающее много строк, обычно задействует меньше ресурсов и выполнится быстрее, но может препятствовать чтению этого материализованного представления другими сеансами. При этом данный режим может быть быстрее при небольшом количестве строк.

Данный параметр допускается, только если в материализованном представлении есть хотя бы один индекс `UNIQUE`, построенный только по именам столбцов и включающий все строки (то есть это не должен быть индекс по выражению или индекс, содержащий `WHERE`).

Этот параметр нельзя использовать, когда материализованное представление ещё не наполнено.

Даже с этим параметром в один момент времени допускается только одно обновление (`REFRESH`) материализованного представления.

имя

Имя (возможно, дополненное схемой) материализованного представления, подлежащего обновлению.

Замечания

Если в запросе, определяющем материализованное представление, есть предложение `ORDER BY`, исходное содержимое материализованного представления будет упорядочено указанным образом; но `REFRESH MATERIALIZED VIEW` не гарантирует сохранения этого порядка.

Примеры

Эта команда заменяет содержимое материализованного представления `order_summary`, используя запрос из определения материализованного представления, и оставляет его в сканируемом состоянии:

```
REFRESH MATERIALIZED VIEW order_summary;
```

Эта команда освобождает пространство, связанное с материализованным представлением `annual_statistics_basis`, и оставляет это представление в несканируемом состоянии:

```
REFRESH MATERIALIZED VIEW annual_statistics_basis WITH NO DATA;
```

Совместимость

`REFRESH MATERIALIZED VIEW` — расширение Postgres Pro.

См. также

CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, DROP MATERIALIZED VIEW

REINDEX

REINDEX — перестроить индексы

Синтаксис

```
REINDEX [ ( VERBOSE ) ] { INDEX | TABLE | SCHEMA | DATABASE | SYSTEM } имя
```

Описание

`REINDEX` перестраивает индекс, обрабатывая данные таблицы, к которой относится индекс, и в результате заменяет старую копию индекса. Команда `REINDEX` применяется в следующих ситуациях:

- Индекс был повреждён, его содержимое стало некорректным. Хотя в теории этого не должно случаться, на практике индексы могут испортиться из-за программных ошибок или аппаратных сбоев. В таких случаях `REINDEX` служит методом восстановления индекса.
- Индекс стал «раздутым», то есть в нём оказалось много пустых или почти пустых страниц. Это может происходить с B-деревьями в Postgres Pro при определённых, достаточно редких сценариях использования. `REINDEX` даёт возможность сократить объём, занимаемый индексом, записывая новую версию индекса без «мёртвых» страниц. За подробностями обратитесь к Разделу 23.2.
- Параметр хранения индекса (например, фактор заполнения) был изменён, и теперь требуется, чтобы это изменение вступило в силу в полной мере.
- Построение индекса с параметром `CONCURRENTLY` завершилось ошибкой, в результате чего индекс оказался «нерабочим». Такие индексы бесполезны, но их можно легко перестроить, воспользовавшись командой `REINDEX`. Однако заметьте, что `REINDEX` будет перестраивать их в обычном, а не в неблокирующем режиме. Чтобы перестроить такой индекс, минимизируя влияние на производственную среду, его следует удалить, а затем снова выполнить команду `CREATE INDEX CONCURRENTLY`.

Параметры

`INDEX`

Перестраивает указанный индекс.

`TABLE`

Перестраивает все индексы в указанной таблице. Если у таблицы имеется дополнительная таблица «TOAST», она так же переиндексируется.

`SCHEMA`

Перестраивает все индексы в указанной схеме. Если таблица в этой схеме имеет вторичную таблицу «TOAST», она также будет переиндексирована. При этом обрабатываются и индексы в общих системных каталогах. Эту форму `REINDEX` нельзя выполнить в блоке транзакции.

`DATABASE`

Перестраивает все индексы в текущей базе данных. При этом обрабатываются также индексы в общих системных каталогах. Эту форму `REINDEX` нельзя выполнить в блоке транзакции.

`SYSTEM`

Перестраивает все индексы в системных каталогах текущей базы данных. При этом обрабатываются также индексы в общих системных каталогах, но индексы в таблицах пользователя не затрагиваются. Эту форму `REINDEX` нельзя выполнить в блоке транзакции.

имя

Имя определённого индекса, таблицы или базы данных, подлежащих переиндексации. В настоящее время `REINDEX DATABASE` и `REINDEX SYSTEM` могут переиндексировать только текущую базу данных, так что их параметр должен соответствовать имени текущей базы данных.

`VERBOSE`

Выводит отчёт о прогрессе после переиндексации каждого индекса.

Замечания

В случае подозрений в повреждении индекса таблицы пользователя, этот индекс или все индексы таблицы можно перестроить, используя команду `REINDEX INDEX` или `REINDEX TABLE`.

Всё усложняется, если возникает необходимость восстановить повреждённый индекс системной таблицы. В этом случае важно, чтобы система сама не использовала этот индекс. (На самом деле в таких случаях вы, скорее всего, столкнётесь с падением процессов сервера в момент запуска, как раз вследствие испорченных индексов.) Чтобы надёжно восстановить рабочее состояние, сервер следует запускать с параметром `-P`, который отключает использование индексов при поиске в системных каталогах.

Один из вариантов сделать это — выключить сервер Postgres Pro и запустить его снова в однопользовательском режиме, с параметром `-P` в командной строке. Затем можно выполнить `REINDEX DATABASE`, `REINDEX SYSTEM`, `REINDEX TABLE` или `REINDEX INDEX`, в зависимости от того, что вы хотите восстановить. В случае сомнений выполните `REINDEX SYSTEM`, чтобы перестроить все системные индексы в базе данных. Затем завершите однопользовательский сеанс сервера и перезапустите сервер в обычном режиме. Чтобы подробнее узнать, как работать с сервером в однопользовательском интерфейсе, обратитесь к справочной странице postgres.

Можно так же запустить обычный экземпляр сервера, но добавить в параметры командной строки `-P`. В разных клиентах это может делаться по-разному, но во всех клиентах на базе libpq можно установить для переменной окружения `PGOPTIONS` значение `-P` до запуска клиента. Учтите, что хотя этот метод не препятствует работе других клиентов, всё же имеет смысл не позволять им подключаться к повреждённой базе данных до завершения восстановления.

Действие `REINDEX` подобно удалению и пересозданию индекса в том смысле, что содержимое индекса пересоздаётся с нуля, но блокировки при этом устанавливаются другие. `REINDEX` блокирует запись, но не чтение родительской таблицы индекса. Эта команда также устанавливает блокировку `ACCESS EXCLUSIVE` для обрабатываемого индекса, что блокирует чтение таблицы, при котором задействуется этот индекс. `DROP INDEX`, напротив, моментально устанавливает блокировку `ACCESS EXCLUSIVE` на родительскую таблицу, блокируя и запись, и чтение. Последующая команда `CREATE INDEX` блокирует запись, но не чтение; так как индекс отсутствует, обращений к нему ни при каком чтении не будет, что означает, что блокироваться чтение не будет, но выполняться оно будет как дорогостоящее последовательное сканирование.

Для перестраивания одного индекса или индексов таблицы необходимо быть владельцем этого индекса или таблицы. Для переиндексирования базы данных необходимо быть владельцем базы данных (заметьте, что он может таким образом перестроить индексы таблиц, принадлежащих другим пользователям). Разумеется, суперпользователи могут переиндексировать всё без ограничений.

Примеры

Перестроение одного индекса:

```
REINDEX INDEX my_index;
```

Перестроение всех индексов таблицы `my_table`:

```
REINDEX TABLE my_table;
```

Перестроение всех индексов в определённой базе данных, в предположении, что целостность системных индексов под сомнением:

```
$ export PGOPTIONS="-P"
$ psql broken_db
...
broken_db=> REINDEX DATABASE broken_db;
broken_db=> \q
```

Совместимость

Команда `REINDEX` отсутствует в стандарте SQL.

RELEASE SAVEPOINT

RELEASE SAVEPOINT — высвободить ранее определённую точку сохранения

Синтаксис

```
RELEASE [ SAVEPOINT ] имя_точки_сохранения
```

Описание

`RELEASE SAVEPOINT` уничтожает точку сохранения, определённую ранее в текущей транзакции.

После уничтожения точка сохранения становится неприменимой в качестве точки возврата, но никаких других проявлений, видимых для пользователя, эта команда не имеет. Она не отменяет эффекта команд, выполненных после установки точки сохранения. (Для этого предназначена команда ROLLBACK TO SAVEPOINT.) Уничтожение точки сохранения, когда она становится не нужна, позволяет системе освобождать некоторые ресурсы раньше, чем завершается транзакция.

`RELEASE SAVEPOINT` также уничтожает все точки сохранения, установленные после заданной точки.

Параметры

имя_точки_сохранения

Имя точки сохранения, подлежащей уничтожению.

Замечания

Указание имени точки сохранения, не определённой ранее, считается ошибкой.

Освободить точку сохранения в транзакции, находящейся в прерванном состоянии, нельзя.

Если одно имя дано нескольким ранее определённым точкам сохранения, освобождена будет только последняя из них. Повторные команды будут последовательно освобождать более ранние точки сохранения.

Примеры

Этот пример показывает, как установить и затем уничтожить точку сохранения:

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

Данная транзакция вставит значения 3 и 4.

Совместимость

Эта команда соответствует стандарту SQL. В стандарте говорится, что ключевое слово `SAVEPOINT` является обязательным, но Postgres Pro позволяет опускать его.

См. также

BEGIN, COMMIT, ROLLBACK, ROLLBACK TO SAVEPOINT, SAVEPOINT

RESET

RESET — восстановить значение по умолчанию заданного параметра времени выполнения

Синтаксис

```
RESET параметр_конфигурации
RESET ALL
```

Описание

`RESET` сбрасывает параметры времени выполнения к значениям по умолчанию. `RESET` — альтернативное написание команды

```
SET параметр_конфигурации TO DEFAULT
```

За подробностями обратитесь к SET.

Значение по умолчанию определяется как значение, которое имел бы этот параметр, если бы для него не выполнялась команда `SET` в текущем сеансе. Фактическое значение может быть задано в компилируемом коде, файле конфигурации, параметрах командной строки или в параметрах по умолчанию для базы данных или пользователя. Если определить его как «значение, которое имеет параметр в начале сеанса», это будет не вполне корректно, так как оно будет сброшено к тому, что задано в файле конфигурации в данный момент. За подробностями обратитесь к Главе 18.

Транзакционное поведение команды `RESET` не отличается от `SET`: её действие будет отменено при откате транзакции.

Параметры

параметр_конфигурации

Имя устанавливаемого параметра конфигурации времени выполнения. Доступные параметры описаны в Главе 18 и на справочной странице SET.

`ALL`

Сбрасывает к значениям по умолчанию все устанавливаемые параметры времени выполнения.

Примеры

Сброс конфигурационной переменной `timezone` к значению по умолчанию:

```
RESET timezone;
```

Совместимость

`RESET` является расширением Postgres Pro.

См. также

SET, SHOW

REVOKE

REVOKE — отозвать права доступа

Синтаксис

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] имя_таблицы [, ...]
         | ALL TABLES IN SCHEMA имя_схемы [, ...] }
    FROM указание_роли [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | REFERENCES } ( имя_столбца [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( имя_столбца [, ...] ) }
    ON [ TABLE ] имя_таблицы [, ...]
    FROM указание_роли [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE имя_последовательности [, ...]
         | ALL SEQUENCES IN SCHEMA имя_схемы [, ...] }
    FROM указание_роли [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE имя_бд [, ...]
    FROM указание_роли [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN имя_домена [, ...]
    FROM указание_роли [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER имя_обёртки_сторонних_данных [, ...]
    FROM указание_роли [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER имя_сервера [, ...]
    FROM указание_роли [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTION имя_функции [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ] [, ...]
```

```
          | ALL FUNCTIONS IN SCHEMA имя_схемы [, ...] }
    FROM указание_роли [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE имя_языка [, ...]
    FROM указание_роли [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT oid_БО [, ...]
    FROM указание_роли [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA имя_схемы [, ...]
    FROM указание_роли [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE табл_пространство [, ...]
    FROM указание_роли [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE имя_типа [, ...]
    FROM указание_роли [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ ADMIN OPTION FOR ]
    имя_роли [, ...] FROM указание_роли [, ...]
    [ GRANTED BY указание_роли ]
    [ CASCADE | RESTRICT ]

Здесь указание_роли:

    [ GROUP ] имя_роли
  | PUBLIC
  | CURRENT_USER
  | SESSION_USER
```

Описание

Команда REVOKE лишает одну или несколько ролей прав, назначенных ранее. Ключевое слово PUBLIC обозначает неявно определённую группу всех ролей.

Различные типы прав подробно рассматриваются в описании команды GRANT.

Заметьте, что любая конкретная роль получает в сумме права, данные непосредственно ей, права, данные любой роли, в которую она включена, а также права, данные группе PUBLIC. Поэтому, например, лишение PUBLIC права SELECT не обязательно будет означать, что все роли лишатся права SELECT для данного объекта: оно сохранится у тех ролей, которым оно дано непосредственно или косвенно, через другую роль. Подобным образом, лишение права SELECT

какого-либо пользователя может не повлиять на его возможность пользоваться правом SELECT, если это право дано группе PUBLIC или другой роли, в которую он включён.

Если указано GRANT OPTION FOR, отзывается только право передачи права, но не само право. Без этого указания отзывается и право, и право распоряжаться им.

Если пользователь обладает правом с правом передачи и он дал его другим пользователям, последнее право считается зависимым. Когда первый пользователь лишается самого права или права передачи и существуют зависимые права, эти зависимые права также отзываются, если дополнительно указано CASCADE; в противном случае операция завершается ошибкой. Это рекурсивное лишение прав затрагивает только права, полученные через цепочку пользователей, которую можно проследить до пользователя, являющегося субъектом команды REVOKE. Таким образом, пользователи могут в итоге сохранить это право, если оно было также получено через других пользователей.

Когда отзывается право доступа к таблице, с ним вместе автоматически отзываются соответствующие права для каждого столбца таблицы (если такие права заданы). С другой стороны, если роли были даны права для таблицы, лишение роли таких же прав на уровне отдельных столбцов ни на что не влияет.

Когда пользователь лишается членства в роли, указание GRANT OPTION меняется на ADMIN OPTION, но в остальном поведение не отличается. Эта форма команды также принимает указание GRANTED BY, но в настоящее время оно игнорируется (проверяется лишь существование заданной в нём роли). Заметьте также, что эта форма команды не принимает избыточное слово GROUP в *указании_роли*.

Замечания

Для просмотра прав, назначенных для существующих таблиц и столбцов, можно воспользоваться командой \dp в psql. Формат её вывода рассматривается в описании GRANT. Для других, не табличных объектов предусмотрены другие команды \d, которые могут показывать в том числе и назначенные для них права.

Пользователь может отзывать только те права, которые он дал другому непосредственно. Если, например, пользователь A дал право с правом передачи пользователю B, а пользователь B, в свою очередь, дал это право пользователю C, то пользователь A не сможет лишить этого права непосредственно C. Вместо этого, пользователь A может лишить права передачи права пользователя B и использовать параметр CASCADE, чтобы этого права по цепочке лишился пользователь C. Или же, например, если и A, и B дали одно и то же право C, то A сможет отозвать право, которое дал он, но не пользователь B, так что в результате C всё равно будет иметь это право.

Если отозвать право доступа к объекту (с помощью REVOKE) попытается не владелец объекта, команда завершится ошибкой, если пользователь не имеет никаких прав для этого объекта. Если же пользователь имеет какие-то права, команда будет выполняться, но пользователь сможет отозвать только те права, которые даны ему с правом распоряжения ими. Формы REVOKE ALL PRIVILEGES будут выдавать предупреждение, если у него вовсе нет таких прав, тогда как другие формы будут выдавать предупреждения, если пользователь не имеет права распоряжаться именно правами, указанными в команде. (В принципе, эти утверждения применимы и к владельцу объекта, но ему разрешено распоряжаться всем правами, поэтому такие ситуации невозможны.)

Если команду GRANT или REVOKE выполняет суперпользователь, эта команда выполняется так, как будто её выполняет владелец затрагиваемого объекта. Так как все права в конце концов исходят от владельца объекта (возможно, косвенно по цепочке или через право распоряжением правом), суперпользователь может отозвать все права, но это может потребовать применения режима CASCADE, как описывалось выше.

REVOKE также может быть выполнена ролью, которая не является владельцем заданного объекта, но является членом роли-владельца, либо членом роли, имеющей права WITH GRANT OPTION для этого объекта. В этом случае команда будет выполнена, как если бы её выполняла содержащая

роль, действительно владеющая объектом или имеющая права WITH GRANT OPTION. Например, если таблица t1 принадлежит роли g1, членом которой является роль u1, то u1 может отзывать права на использование t1, которые записаны как данные ролью g1. В том числе это могут быть права, данные ролью u1, а также другими членами роли g1.

Если роль, выполняющая команду REVOKE, получила указанные права косвенно по нескольким путям членства ролей, какая именно роль будет выбрана для выполнения команды, не определено. В таких случаях рекомендуется воспользоваться командой SET ROLE и переключиться на роль, которую хочется видеть в качестве выполняющей REVOKE. Если этого не сделать, могут быть отозваны не те права, что планировалось, либо отозвать права вообще не удастся.

Примеры

Лишение группы public права добавлять данные в таблицу films:

```
REVOKE INSERT ON films FROM PUBLIC;
```

Лишение пользователя manuel всех прав для представления kinds:

```
REVOKE ALL PRIVILEGES ON kinds FROM manuel;
```

Заметьте, что на самом деле это означает «лишить всех прав, которые дал я».

Исключение из членов роли admins пользователя joe:

```
REVOKE admins FROM joe;
```

Совместимость

Замечания по совместимости, приведённые для команды GRANT, справедливы и для REVOKE. Стандарт требует обязательного указания ключевого слова RESTRICT или CASCADE, но Postgres Pro подразумевает RESTRICT по умолчанию.

См. также

GRANT

ROLLBACK

ROLLBACK — прервать текущую транзакцию

Синтаксис

```
ROLLBACK [ WORK | TRANSACTION ]
```

Описание

`ROLLBACK` откатывает текущую транзакцию и приводит к аннулированию всех изменений, произведённых транзакцией.

Параметры

`WORK`
`TRANSACTION`

Необязательные ключевые слова, не оказывают никакого влияния.

Замечания

Чтобы завершить и зафиксировать транзакцию, используйте COMMIT.

При выполнении команды `ROLLBACK` вне блока транзакции выдаётся предупреждение и больше ничего не происходит.

Примеры

Чтобы прервать все операции:

```
ROLLBACK;
```

Совместимость

В стандарте SQL описаны только две формы: `ROLLBACK` и `ROLLBACK WORK`. В остальном эта команда полностью соответствует стандарту.

См. также

BEGIN, COMMIT, ROLLBACK TO SAVEPOINT

ROLLBACK PREPARED

ROLLBACK PREPARED — отменить транзакцию, которая ранее была подготовлена для двухфазной фиксации

Синтаксис

```
ROLLBACK PREPARED id_транзакции
```

Описание

`ROLLBACK PREPARED` откатывает транзакцию в подготовленном состоянии.

Параметры

id_транзакции

Идентификатор транзакции, которую нужно откатить.

Замечания

Откатить подготовленную транзакцию может либо пользователь, выполнявший её изначально, либо суперпользователь. При этом не обязательно работать в том же сеансе, где выполнялась транзакция.

Эту команду нельзя выполнить внутри блока транзакции. Подготовленная транзакция откатывается немедленно.

Все существующие в текущий момент подготовленные транзакции показываются в системном представлении `pg_prepared_xacts`.

Примеры

Откат транзакции, имеющей идентификатор `foobar`:

```
ROLLBACK PREPARED 'foobar';
```

Совместимость

Оператор `ROLLBACK PREPARED` является расширением Postgres Pro. Он предназначен для использования внешними системами управления транзакциями, некоторые из которых работают по стандартам (например, X/Open XA), но сторона SQL в этих системах не стандартизирована.

См. также

PREPARE TRANSACTION, COMMIT PREPARED

ROLLBACK TO SAVEPOINT

ROLLBACK TO SAVEPOINT — откатиться к точке сохранения

Синтаксис

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] имя_точки_сохранения
```

Описание

Откатывает все команды, выполненные после установления точки сохранения. Точка сохранения остаётся действующей и при необходимости можно снова откатиться к ней позже.

`ROLLBACK TO SAVEPOINT` неявно уничтожает все точки сохранения, установленные после заданной точки.

Параметры

имя_точки_сохранения

Точка сохранения, к которой нужно откатиться.

Замечания

Чтобы уничтожить точку сохранения, не отменяя действия команд, выполненных после неё, применяется команда RELEASE SAVEPOINT.

Указание имени точки сохранения, не установленной ранее, считается ошибкой.

Курсоры проявляют не совсем транзакционное поведение применительно к точкам сохранения. Любой курсор, открытый внутри точки сохранения, будет закрыт при откате к этой точке. Если ранее открытый курсор был перемещён командой `FETCH` или `MOVE` внутри точки сохранения, к которой затем произошёл откат, курсор остаётся в той позиции, в которой он остался после `FETCH` (то есть, перемещение курсора, производимое командой `FETCH`, не откатывается). Также при откате не отменяется и закрытие курсора. Однако другие побочные эффекты, вызываемые запросом курсора (например, побочные действия изменчивых функций, вызываемых в запросе) *отменяются*, если они производятся после точки сохранения, к которой затем происходит откат. Курсор, выполнение которого приводит к прерыванию транзакции, переводится в нерабочее состояние, так что даже если восстановить транзакцию, выполнив `ROLLBACK TO SAVEPOINT`, этот курсор нельзя будет использовать.

Примеры

Отмена действия команд, выполненных после установки точки сохранения `my_savepoint`:

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

Откат к точке сохранения не отражается на положении курсора:

```
BEGIN;

DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;

SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        1
```

```
ROLLBACK TO SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        2

COMMIT;
```

Совместимость

В стандарте SQL говорится, что ключевое слово `SAVEPOINT` является обязательным, но Postgres Pro и Oracle позволяют опускать его. SQL допускает `WORK`, но не `TRANSACTION`, в качестве избыточного слова после `ROLLBACK`. Кроме того, в SQL есть дополнительное предложение `AND [ NO ] CHAIN`, которое в настоящее время не поддерживается в Postgres Pro. В остальном эта команда соответствует стандарту SQL.

См. также

BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, SAVEPOINT

SAVEPOINT

SAVEPOINT — определить новую точку сохранения в текущей транзакции

Синтаксис

```
SAVEPOINT имя_точки_сохранения
```

Описание

`SAVEPOINT` устанавливает новую точку сохранения в текущей транзакции.

Точка сохранения — это специальная отметка внутри транзакции, которая позволяет откатить все команды, выполненные после неё, и восстановить таким образом состояние на момент установки этой точки.

Параметры

имя_точки_сохранения

Имя, назначаемое новой точке сохранения. Ранее существующая точка с таким именем оказывается недоступной, пока не будут освобождены установленные позже одноимённые точки.

Замечания

Для отката к установленной точке сохранения предназначена команда ROLLBACK TO SAVEPOINT. Чтобы уничтожить точку сохранения, сохраняя изменения, произведённые после того, как она была установлена, применяется команда RELEASE SAVEPOINT.

Точки сохранения могут быть установлены только внутри блока транзакции. В одной транзакции можно определить несколько точек сохранения.

Примеры

Установление точки сохранения и затем отмена действия всех команд, выполненных после установленной точки:

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

Показанная транзакция вставит в таблицу значения 1 и 3, но не 2.

Этот пример показывает, как установить и затем уничтожить точку сохранения:

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

Данная транзакция вставит значения 3 и 4.

Этот пример показывает, как использовать точки сохранения с одним именем:

```
BEGIN;
```

```
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);

    -- откат ко второй точке сохранения
    ROLLBACK TO SAVEPOINT my_savepoint;
    SELECT * FROM table1;               -- вывод строк 1 и 2

    -- освобождение второй точки сохранения
    RELEASE SAVEPOINT my_savepoint;

    -- откат к первой точке сохранения
    ROLLBACK TO SAVEPOINT my_savepoint;
    SELECT * FROM table1;               -- вывод только строки 1
COMMIT;
```

Данная транзакция сначала откатит строку 3, затем строку 2.

Совместимость

Стандарт SQL требует, чтобы точка сохранения уничтожалась автоматически, когда устанавливается другая точка сохранения с тем же именем. В Postgres Pro старая точка сохранения остаётся, хотя при откате или уничтожении будет выбираться только самая последняя. (После уничтожения последней точки командой `RELEASE SAVEPOINT` доступной для команд `ROLLBACK TO SAVEPOINT` и `RELEASE SAVEPOINT` становится следующая.) В остальном оператор `SAVEPOINT` полностью соответствует стандарту.

См. также

BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, ROLLBACK TO SAVEPOINT

SECURITY LABEL

SECURITY LABEL — определить или изменить метку безопасности, применённую к объекту

Синтаксис

```
SECURITY LABEL [ FOR провайдер ] ON
{
  TABLE имя_объекта |
  COLUMN имя_таблицы.имя_столбца |
  AGGREGATE имя_агрегатной_функции ( сигнатура_агр_функции ) |
  DATABASE имя_объекта |
  DOMAIN имя_объекта |
  EVENT TRIGGER имя_объекта |
  FOREIGN TABLE имя_объекта
  FUNCTION имя_функции [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ] |
  LARGE OBJECT oid_большого_объекта |
  MATERIALIZED VIEW имя_объекта |
  [ PROCEDURAL ] LANGUAGE имя_объекта |
  PUBLICATION имя_объекта |
  ROLE имя_объекта |
  SCHEMA имя_объекта |
  SEQUENCE имя_объекта |
  SUBSCRIPTION имя_объекта |
  TABLESPACE имя_объекта |
  TYPE имя_объекта |
  VIEW имя_объекта
} IS 'метка'

Здесь сигнатура_агр_функции:

* |
[ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] |
[ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] ] ORDER BY
 [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ]
```

Описание

`SECURITY LABEL` применяет метку безопасности к объекту базы данных. С определённым объектом может быть связано произвольное количество меток безопасности, по одной для каждого провайдера. Провайдеры меток представляют собой загружаемые модули, которые регистрируют себя, вызывая функцию `register_label_provider`.

> **Примечание**
>
> `register_label_provider` — это не SQL-функция; её можно вызывать только из скомпилированного кода C, загруженного сервером.

Провайдер меток определяет, допустима ли заданная метка и разрешено ли применять эту метку к указанному объекту. Какой смысл вкладывается в данную метку, тоже определяет провайдер меток. Postgres Pro не накладывает никаких ограничений на то, как провайдер должен интерпретировать метки безопасности; он просто обеспечивает механизм их хранения. На практике, этот механизм реализован для того, чтобы в базы данных можно было интегрировать системы мандатного управления доступом (MAC) на базе меток, такие как SELinux. Такие системы

принимают все решения по ограничению доступа, учитывая метки объектов, а не традиционные сущности избирательного управления доступом (DAC), такие как пользователи и группы.

Параметры

имя_объекта
имя_таблицы.имя_столбца
имя_агрегатной_функции
имя_функции

Имя помечаемого объекта. Имена таблиц, агрегатных и обычных функций, доменов, сторонних таблиц, последовательностей и представлений можно дополнить именем схемы.

провайдер

Имя провайдера, с которым будет связана эта метка. Указанный провайдер должен быть загружен и готов выполнять операцию размечивания. Если загружен всего один провайдер, его имя можно опустить для краткости.

режим_аргумента

Режим аргумента обычной или агрегатной функции: `IN`, `OUT`, `INOUT` или `VARIADIC`. По умолчанию подразумевается `IN`. Заметьте, что `SECURITY LABEL` не учитывает аргументы `OUT`, так как для идентификации функции нужны только типы входных аргументов. Поэтому достаточно перечислить только аргументы `IN`, `INOUT` и `VARIADIC`.

имя_аргумента

Имя аргумента обычной или агрегатной функции. Заметьте, что на самом деле `SECURITY LABEL` не обращает внимание на имена аргументов, так как для однозначной идентификации функции достаточно только типов аргументов.

тип_аргумента

Тип данных аргумента обычной или агрегатной функции.

oid_большого_объекта

OID большого объекта.

`PROCEDURAL`

Это слово не несёт смысловой нагрузки.

метка

Новая метка безопасности, записанная в виде строковой константы, либо `NULL`, если метку безопасности нужно удалить.

Примеры

Следующий пример показывает, как можно изменить метку безопасности для таблицы.

```
SECURITY LABEL FOR selinux ON TABLE mytable IS 'system_u:object_r:sepgsql_table_t:s0';
```

Совместимость

Команда `SECURITY LABEL` отсутствует в стандарте SQL.

См. также

sepgsql, `src/test/modules/dummy_seclabel`

SELECT

SELECT, TABLE, WITH — получить строки из таблицы или представления

Синтаксис

```
[ WITH [ RECURSIVE ] запрос_WITH [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( выражение [, ...] ) ] ]
    [ * | выражение [ [ AS ] имя_результата ] [, ...] ]
    [ FROM элемент_FROM [, ...] ]
    [ WHERE условие ]
    [ GROUP BY элемент_группирования [, ...] ]
    [ HAVING условие ]
    [ WINDOW имя_окна AS ( определение_окна ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] выборка ]
    [ ORDER BY выражение [ ASC | DESC | USING оператор ] [ NULLS { FIRST | LAST } ]
 [, ...] ]
    [ LIMIT { число | ALL } ]
    [ OFFSET начало [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ число ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF имя_таблицы [, ...] ]
 [ NOWAIT | SKIP LOCKED ] [...] ]

Здесь допускается элемент_FROM:

    [ ONLY ] имя_таблицы [ * ] [ [ AS ] псевдоним [ ( псевдоним_столбца [, ...] ) ] ]
                [ TABLESAMPLE метод_выборки ( аргумент [, ...] ) [ REPEATABLE
 ( затравка ) ] ]
    [ LATERAL ] ( выборка ) [ AS ] псевдоним [ ( псевдоним_столбца [, ...] ) ]
    имя_запроса_WITH [ [ AS ] псевдоним [ ( псевдоним_столбца [, ...] ) ] ]
    [ LATERAL ] имя_функции ( [ аргумент [, ...] ] )
                [ WITH ORDINALITY ] [ [ AS ] псевдоним [ ( псевдоним_столбца
 [, ...] ) ] ]
    [ LATERAL ] имя_функции ( [ аргумент [, ...] ] ) [ AS ] псевдоним
 ( определение_столбца [, ...] )
    [ LATERAL ] имя_функции ( [ аргумент [, ...] ] ) AS ( определение_столбца [, ...] )
    [ LATERAL ] ROWS FROM( имя_функции ( [ аргумент [, ...] ] ) [ AS
 ( определение_столбца [, ...] ) ] [, ...] )
                [ WITH ORDINALITY ] [ [ AS ] псевдоним [ ( псевдоним_столбца
 [, ...] ) ] ]
    элемент_FROM [ NATURAL ] тип_соединения элемент_FROM [ ON условие_соединения |
 USING ( столбец_соединения [, ...] ) ]

и элемент_группирования может быть следующим:

    ( )
    выражение
    ( выражение [, ...] )
    ROLLUP ( { выражение | ( выражение [, ...] ) } [, ...] )
    CUBE ( { выражение | ( выражение [, ...] ) } [, ...] )
    GROUPING SETS ( элемент_группирования [, ...] )

и запрос_WITH:

    имя_запроса_WITH [ ( имя_столбца [, ...] ) ] AS ( выборка | values | insert
 | update | delete )
```

```
TABLE [ ONLY ] имя_таблицы [ * ]
```

Описание

`SELECT` получает строки из множества таблиц (возможно, пустого). Общая процедура выполнения `SELECT` следующая:

1. Выполняются все запросы в списке `WITH`. По сути они формируют временные таблицы, к которым затем можно обращаться в списке `FROM`. Запрос в `WITH` выполняется только один раз, даже если он фигурирует в списке `FROM` неоднократно. (См. Подраздел «Предложение `WITH`» ниже.)
2. Вычисляются все элементы в списке `FROM`. (Каждый элемент в списке `FROM` представляет собой реальную или виртуальную таблицу.) Если список `FROM` содержит несколько элементов, они объединяются перекрёстным соединением. (См. Подраздел «Предложение `FROM`» ниже.)
3. Если указано предложение `WHERE`, все строки, не удовлетворяющие условию, исключаются из результата. (См. Подраздел «Предложение `WHERE`» ниже.)
4. Если присутствует указание `GROUP BY`, либо в запросе вызываются агрегатные функции, вывод разделяется по группам строк, соответствующим одному или нескольким значениям, а затем вычисляются результаты агрегатных функций. Если добавлено предложение `HAVING`, оно исключает группы, не удовлетворяющие заданному условию. (См. Подраздел «Предложение `GROUP BY`» и Подраздел «Предложение `HAVING`» ниже.)
5. Вычисляются фактические выходные строки по заданным в `SELECT` выражениям для каждой выбранной строки или группы строк. (См. Подраздел «Список `SELECT`» ниже.)
6. `SELECT DISTINCT` исключает из результата повторяющиеся строки. `SELECT DISTINCT ON` исключает строки, совпадающие по всем указанным выражениям. `SELECT ALL` (по умолчанию) возвращает все строки результата, включая дубликаты. (См. Подраздел «Предложение `DISTINCT`» ниже.)
7. Операторы `UNION`, `INTERSECT` и `EXCEPT` объединяют вывод нескольких команд `SELECT` в один результирующий набор. Оператор `UNION` возвращает все строки, представленные в одном, либо обоих наборах результатов. Оператор `INTERSECT` возвращает все строки, представленные строго в обоих наборах. Оператор `EXCEPT` возвращает все строки, представленные в первом наборе, но не во втором. Во всех трёх случаях повторяющиеся строки исключаются из результата, если явно не указано `ALL`. Чтобы явно обозначить, что выдаваться должны только неповторяющиеся строки, можно добавить избыточное слово `DISTINCT`. Заметьте, что в данном контексте по умолчанию подразумевается `DISTINCT`, хотя в самом `SELECT` по умолчанию подразумевается `ALL`. (См. Подраздел «Предложение `UNION`+», Подраздел «Предложение `INTERSECT`» и Подраздел «Предложение `EXCEPT`» ниже.)
8. Если присутствует предложение `ORDER BY`, возвращаемые строки сортируются в указанном порядке. В отсутствие `ORDER BY` строки возвращаются в том порядке, в каком системе будет проще их выдать. (См. Подраздел «Предложение `ORDER BY`» ниже.)
9. Если указано предложение `LIMIT` (или `FETCH FIRST`) либо `OFFSET`, оператор `SELECT` возвращает только подмножество строк результата. (См. Подраздел «Предложение `LIMIT`» ниже.)
10. Если указано `FOR UPDATE`, `FOR NO KEY UPDATE`, `FOR SHARE` или `FOR KEY SHARE`, оператор `SELECT` блокирует выбранные строки, защищая их от одновременных изменений. (См. Подраздел «Предложение блокировки» ниже.)

Для всех столбцов, задействованных в команде `SELECT`, необходимо иметь право `SELECT`. Применение блокировок `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE` или `FOR KEY SHARE` требует также права `UPDATE` (как минимум для одного столбца в каждой выбранной для блокировки таблице).

Параметры

Предложение `WITH`

Предложение `WITH` позволяет задать один или несколько подзапросов, к которым затем можно обратиться по имени в основном запросе. Эти подзапросы по сути действуют как временные

таблицы или представления в процессе выполнения главного запроса. Каждый подзапрос может представлять собой оператор SELECT, TABLE, VALUES, INSERT, UPDATE или DELETE. При использовании в WITH оператора, изменяющего данные, (INSERT, UPDATE или DELETE) обычно добавляется предложение RETURNING. Заметьте, что именно результат RETURNING, а *не* нижележащая таблица, изменяемая запросом, формирует временную таблицу, которую затем читает основной запрос. Если RETURNING опущено, оператор тем не менее выполняется, но не выдаёт никакого результата, так что на него нельзя сослаться как на таблицу в основном запросе.

Имя (без схемы) должно быть указано для каждого запроса WITH. Также можно задать необязательный список с именами столбцов; если он опущен, имена столбцов формируются из результата подзапроса.

Если указано RECURSIVE, подзапрос SELECT может ссылаться сам на себя по имени. Такой подзапрос должен иметь форму

нерекурсивная_часть UNION [ALL | DISTINCT] *рекурсивная_часть*

, где рекурсивная ссылка на сам запрос может находиться только справа от UNION. Для одного запроса допускается только одна рекурсивная ссылка на него же. Операторы, изменяющие данные, не могут быть рекурсивными, но результат рекурсивного запроса SELECT в таких операторах можно использовать. За примером обратитесь к Разделу 7.8.

Ещё одна особенность RECURSIVE в том, что запросы WITH могут быть неупорядоченными: запрос может ссылаться на другой, идущий в списке после него. (Однако циклические ссылки или взаимная рекурсия не поддерживаются.) Без RECURSIVE запрос в WITH может ссылаться только на запросы того же уровня в WITH, предшествующие ему в списке WITH.

Ключевое свойство запросов WITH состоит в том, что они вычисляются один раз для всего основного запроса, даже если в основном запросе содержатся несколько ссылок на них. В частности, гарантируется, что операторы, изменяющие данные, будут выполняться ровно один раз, вне зависимости от того, будет ли их результат прочитан основным запросом и в каком объёме.

Когда в предложении WITH задаются несколько запросов, RECURSIVE следует указывать только единожды, сразу после WITH. Это указание будет действовать на все запросы в предложении WITH, хотя оно никак не скажется на запросах, не использующих рекурсию или ссылки на последующие запросы.

Основной запрос и все запросы WITH, условно говоря, выполняются одновременно. Это значит, что действие оператора, изменяющего данные в WITH, не будут видеть другие части запроса, кроме как прочитав его вывод RETURNING. Если два таких оператора попытаются изменить одну строку, результат будет неопределённым.

За дополнительными сведениями обратитесь к Разделу 7.8.

Предложение FROM

В предложении FROM перечисляются одна или несколько таблиц, служащих источниками данных для SELECT. Если указано несколько источников, результатом будет декартово произведение (перекрёстное соединение) всех их строк. Но обычно в запрос добавляются уточняющие условия (в предложении WHERE), которые ограничивают набор строк небольшим подмножеством этого произведения.

Предложение FROM может содержать следующие элементы:

имя_таблицы

Имя (возможно, дополненное схемой) существующей таблицы или представления. Если перед именем таблицы указано ONLY, считывается только заданная таблица. Без ONLY считывается и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно также добавить необязательное указание *, чтобы явно обозначить, что блокировка затрагивает и все дочерние таблицы.

псевдоним

Альтернативное имя для элемента списка FROM. Этот псевдоним используется для краткости или для исключения неоднозначности с замкнутыми соединениями (когда одна таблица читается неоднократно). Когда задаётся псевдоним, он полностью скрывает настоящее имя таблицы или функции; например, при записи FROM foo AS f, в продолжении запроса SELECT к этому элементу FROM нужно обращаться по имени f, а не foo. Если задан псевдоним таблицы, за ним можно также написать список псевдонимов столбцов, который определит альтернативные имена для столбцов таблицы.

TABLESAMPLE *метод_выборки* (*аргумент* [, ...]) [REPEATABLE (*затравка*)]

Предложение TABLESAMPLE, сопровождающее *имя_таблицы*, показывает, что для получения подмножества строк в этой таблице должен применяться указанный *метод_выборки*. Эта выборка предшествует применению любых других фильтров, например, в предложении WHERE. В стандартный дистрибутив Postgres Pro включены два метода выборки, BERNOULLI и SYSTEM; другие методы выборки можно установить в базу данных через расширения.

Методы выборки BERNOULLI и SYSTEM принимают единственный *аргумент*, определяющий, какой процент таблицы должен попасть в выборку, от 0 до 100. Этот аргумент может задаваться любым выражением со значением типа real. (Другие методы выборки могут принимать дополнительные или другие параметры.) Оба этих метода возвращают случайную выборку таблицы, содержащую примерно указанный процент строк таблицы. Метод BERNOULLI сканирует всю таблицу и выбирает или игнорирует отдельные строки независимо, с заданной вероятностью. Метод SYSTEM строит выборку на уровне блоков, определяя для каждого блока шанс его задействовать, и возвращает все строки из каждого задействуемого блока. Метод SYSTEM работает значительно быстрее BERNOULLI, когда выбирается небольшой процент строк, но он может выдавать менее случайную выборку таблицы из-за эффектов кучности.

В необязательном предложении REPEATABLE задаётся *затравка* — число или выражение, задающее отправное значение для генератора случайных чисел в методе выборки. Значением затравки может быть любое отличное от NULL число с плавающей точкой. Два запроса, в которых указаны одинаковые значения затравки и *аргумента*, выдадут одну и ту же выборку таблицы при условии неизменности содержимого таблицы. Но с разными значениями затравки выборки обычно получаются разными. В отсутствие предложения REPEATABLE для каждого запроса выдаётся новая случайная выборка, в зависимости от затравки, сгенерированной системой. Заметьте, что некоторые дополнительные методы выборки не принимают предложение REPEATABLE и выдают разные выборки при каждом использовании.

выборка

Предложение FROM может содержать вложенный запрос SELECT. Можно считать, что из его результата создаётся временная таблица на время выполнения основной команды SELECT. Заметьте, что вложенный запрос SELECT должен заключаться в скобки и для него *должен* задаваться псевдоним. Здесь также можно использовать команду VALUES.

имя_запроса_WITH

На запрос WITH можно ссылаться по имени, как если бы имя запроса представляло имя таблицы. (На самом деле запрос WITH скрывает любую реальную таблицу с тем же именем для основного запроса. Если необходимо обратиться к одноимённой реальной таблице, можно дополнить имя этой таблицы именем схемы.) Для этого имени можно задать псевдоним, так же, как и для имени таблицы.

имя_функции

В предложении FROM могут содержаться вызовы функций. (Это особенно полезно для функций, возвращающих множества, но в принципе можно использовать любые функции.) Можно считать, что из результата функции создаётся временная таблица на время выполнения основной команды SELECT. Если вызов функции сопровождается необязательным

предложением WITH ORDINALITY, после всех выдаваемых функцией столбцов в вывод добавляется ещё один столбец с номерами строк.

Псевдоним для функции можно задать так же, как и для таблицы. Если этот псевдоним задан, за ним можно также написать список псевдонимов столбцов, который определит альтернативные имена для атрибутов составного типа результата функции, включая имя столбца, который может быть добавлен предложением ORDINALITY.

Несколько вызовов функций можно объединить в одном элементе предложения FROM, заключив их в конструкцию ROWS FROM(...). Выводом такого элемента будет соединение первых строк всех функций, затем вторых строк и т. д. Если одни функции выдают меньше строк, чем другие, недостающие данные заменяются значениями NULL, так что общее число возвращаемых строк всегда будет равняться максимальному числу строк из возвращённых всеми функциями.

Если функция определена как возвращающая тип данных record, для неё нужно указать псевдоним или ключевое слово AS, за которым должен идти список определений столбцов в форме (*имя_столбца тип_данных* [, ...]). Список определений столбцов должен соответствовать фактическому количеству и типу столбцов, возвращаемых функцией.

Если при использовании синтаксиса ROWS FROM(...) одна из функций требует наличия списка определений столбцов, этот список лучше разместить после вызова функции внутри ROWS FROM(...). Список определений столбцов можно поместить после конструкции ROWS FROM(...), только если вызывается всего одна функция, а предложение WITH ORDINALITY отсутствует.

Чтобы использовать ORDINALITY со списком определений столбцов, необходимо применить запись ROWS FROM(...) и поместить список с определениями столбцов внутрь ROWS FROM(...).

тип_соединения

Один из следующих вариантов:

- [INNER] JOIN
- LEFT [OUTER] JOIN
- RIGHT [OUTER] JOIN
- FULL [OUTER] JOIN
- CROSS JOIN

Для типов соединений INNER и OUTER необходимо указать условие соединения, а именно одно из предложений NATURAL, ON *условие_соединения* или USING (*столбец_соединения* [, ...]). Эти предложения описываются ниже. Для CROSS JOIN ни одно из этих предложений не допускается.

Предложение JOIN объединяет два элемента списка FROM, которые мы для простоты дальше будем называть «таблицами», хотя на самом деле это может быть любой объект, допустимый в качестве элемента FROM. Для определения порядка вложенности при необходимости следует использовать скобки. В отсутствие скобок предложения JOIN обрабатывается слева направо. В любом случае JOIN связывает элементы сильнее, чем запятые, разделяющие элементы в списке FROM.

CROSS JOIN и INNER JOIN формируют простое декартово произведение, то же, что можно получить, указав две таблицы на верхнем уровне FROM, но ограниченное возможным условием соединения. Предложение CROSS JOIN равнозначно INNER JOIN ON (TRUE), то есть, никакие строки по условию не удаляются. Эти типы соединений введены исключительно для удобства записи, они не дают ничего такого, что нельзя было бы получить, используя просто FROM и WHERE.

LEFT OUTER JOIN возвращает все строки ограниченного декартова произведения (т. е. все объединённые строки, удовлетворяющие условию соединения) плюс все строки в таблице

слева, для которых не находится строк в таблице справа, удовлетворяющих условию. Строка, взятая из таблицы слева, дополняется до полной ширины объединённой таблицы значениями NULL в столбцах таблицы справа. Заметьте, что для определения, какие строки двух таблиц соответствуют друг другу, проверяется только условие самого предложения JOIN. Внешние условия проверяются позже.

RIGHT OUTER JOIN, напротив, возвращает все соединённые строки плюс одну строку для каждой строки справа, не имеющей соответствия слева (эта строка дополняется значениями NULL влево). Это предложение введено исключительно для удобства записи, так как его можно легко свести к LEFT OUTER JOIN, поменяв левую и правую таблицы местами.

FULL OUTER JOIN возвращает все соединённые строки плюс все строки слева, не имеющие соответствия справа, (дополненные значениями NULL вправо) плюс все строки справа, не имеющие соответствия слева (дополненные значениями NULL влево).

ON *условие_соединения*

Задаваемое *условие_соединения* представляет собой выражение, выдающее значение типа boolean (как в предложении WHERE), которое определяет, какие строки считаются соответствующими при соединении.

USING (*столбец_соединения* [, ...])

Предложение вида USING (a, b, ...) представляет собой сокращённую форму записи ON таблица_слева.a = таблица_справа.a AND таблица_слева.b = таблица_справа.b .... Кроме того, USING подразумевает, что в результат соединения будет включён только один из пары равных столбцов, но не оба.

NATURAL

NATURAL представляет собой краткую запись USING со списком, в котором перечисляются все столбцы двух таблиц, имеющие одинаковые имена. Если одинаковых имён нет, указание NATURAL равнозначно ON TRUE.

LATERAL

Ключевое слово LATERAL может предварять вложенный запрос SELECT в списке FROM. Оно позволяет обращаться в этом вложенном SELECT к столбцам элементов FROM, предшествующим ему в списке FROM. (Без LATERAL все вложенные подзапросы SELECT обрабатываются независимо и не могут ссылаться на другие элементы списка FROM.)

Слово LATERAL можно также добавить перед вызовом функции в списке FROM, но в этом случае оно будет избыточным, так как выражения с функциями могут ссылаться на предыдущие элементы списка FROM в любом случае.

Элемент LATERAL может находиться на верхнем уровне списка FROM или в дереве JOIN. В последнем случае он может также ссылаться на любые элементы в левой части JOIN, справа от которого он находится.

Когда элемент FROM содержит ссылки LATERAL, запрос выполняется следующим образом: сначала для строки элемента FROM с целевыми столбцами, или набора строк из нескольких элементов FROM, содержащих целевые столбцы, вычисляется элемент LATERAL со значениями этих столбцов. Затем результирующие строки обычным образом соединяются со строками, из которых они были вычислены. Эта процедура повторяется для всех строк исходных таблиц.

Таблица, служащая источником столбцов, должна быть связана с элементом LATERAL соединением INNER или LEFT, в противном случае не образуется однозначно определяемый набор строк, из которого можно будет получать наборы строк для элемента LATERAL. Таким образом, хотя конструкция *X* RIGHT JOIN LATERAL *Y* синтаксически правильная, *Y* в ней не может обращаться к *X*.

Предложение WHERE

Необязательное предложение WHERE имеет общую форму

```
WHERE условие
```

, где *условие* — любое выражение, выдающее результат типа boolean. Любая строка, не удовлетворяющая этому условию, исключается из результата. Строка удовлетворяет условию, если оно возвращает true при подстановке вместо ссылок на переменные фактических значений из этой строки.

Предложение GROUP BY

Необязательное предложение GROUP BY имеет общую форму

```
GROUP BY элемент_группирования [, ...]
```

GROUP BY собирает в одну строку все выбранные строки, выдающие одинаковые значения для выражений группировки. В качестве *выражения* внутри *элемента_группирования* может выступать имя входного столбца, либо имя или порядковый номер выходного столбца (из списка элементов SELECT), либо произвольное значение, вычисляемое по значениям входных столбцов. В случае неоднозначности имя в GROUP BY будет восприниматься как имя входного, а не выходного столбца.

Если в элементе группирования задаётся GROUPING SETS, ROLLUP или CUBE, предложение GROUP BY в целом определяет некоторое число независимых *наборов группирования*. Это даёт тот же эффект, что и объединение подзапросов (с UNION ALL) с отдельными наборами группирования в их предложениях GROUP BY. Подробнее использование наборов группирования описывается в Подразделе 7.2.4.

Агрегатные функции, если они используются, вычисляются по всем строкам, составляющим каждую группу, и в итоге выдают отдельное значение для каждой группы. (Если агрегатные функции используются без предложения GROUP BY, запрос выполняется как с одной группой, включающей все выбранные строки.) Набор строк, поступающих в каждую агрегатную функцию, можно дополнительно отфильтровать, добавив предложение FILTER к вызову агрегатной функции; за дополнительными сведениями обратитесь к Подразделу 4.2.7. С предложением FILTER на вход агрегатной функции поступают только те строки, которые соответствуют заданному фильтру.

Когда в запросе присутствует предложение GROUP BY или какая-либо агрегатная функция, выражения в списке SELECT не могут обращаться к негруппируемым столбцам, кроме как в агрегатных функциях или в случае функциональной зависимости, так как иначе в негруппируемом столбце нужно было бы вернуть более одного возможного значения. Функциональная зависимость образуется, если группируемые столбцы (или их подмножество) составляют первичный ключ таблицы, содержащей негруппируемый столбец.

Имейте в виду, что все агрегатные функции вычисляются перед «скалярными» выражениями в предложении HAVING или списке SELECT. Это значит, что например, с помощью выражения CASE нельзя обойти вычисление агрегатной функции; см. Подраздел 4.2.14.

В настоящее время указания FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE нельзя задать вместе с GROUP BY.

Предложение HAVING

Необязательное предложение HAVING имеет общую форму

```
HAVING условие
```

Здесь *условие* задаётся так же, как и для предложения WHERE.

HAVING исключает из результата строки групп, не удовлетворяющих условию. HAVING отличается от WHERE: WHERE фильтрует отдельные строки до применения GROUP BY, а HAVING фильтрует строки групп, созданных предложением GROUP BY. Каждый столбец, фигурирующий в *условии*, должен однозначно ссылаться на группируемый столбец, за исключением случаев, когда эта ссылка

находится внутри агрегатной функции или негруппируемый столбец функционально зависит от группируемых.

В присутствие HAVING запрос превращается в группируемый, даже если GROUP BY отсутствует. То же самое происходит, когда запрос содержит агрегатные функции, но не предложение GROUP BY. Все выбранные строки считаются формирующими одну группу, а в списке SELECT и предложении HAVING можно обращаться к столбцам таблицы только из агрегатных функций. Такой запрос будет выдавать единственную строку, если результат условия HAVING — true, и ноль строк в противном случае.

В настоящее время указания FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE нельзя задать вместе с HAVING.

Предложение WINDOW

Необязательное предложение WINDOW имеет общую форму

```
WINDOW имя_окна AS ( определение_окна ) [, ...]
```

Здесь *имя_окна* — это имя, на которое можно ссылаться из предложений OVER или последующих определений окон, а *определение_окна* имеет следующий вид:

```
[ имя_существующего_окна ]
[ PARTITION BY выражение [, ...] ]
[ ORDER BY выражение [ ASC | DESC | USING оператор ] [ NULLS { FIRST | LAST } ]
 [, ...] ]
[ предложение_рамки ]
```

Если указано *имя_существующего_окна*, оно должно ссылаться на предшествующую запись в списке WINDOW; новое окно копирует предложение разбиения из этой записи, а также предложение сортировки, если оно присутствует. В этом случае для нового окна нельзя задать собственное предложение PARTITION BY, а ORDER BY можно указать, только если его не было у копируемого окна. Новое окно всегда использует собственное предложение рамки; в копируемом окне оно задаваться не должно.

Элементы списка PARTITION BY интерпретируется во многом так же, как и элементы Подраздела «Предложение GROUP BY», за исключением того, что это всегда простые выражения, но не имя или номер выходного столбца. Другое различие состоит в том, что эти выражения могут содержать вызовы агрегатных функций, которые не допускаются в обычном предложении GROUP BY. Здесь они допускаются потому, что формирование окна происходит после группировки и агрегирования.

Подобным образом, элементы списка ORDER BY интерпретируются во многом так же, как и элементы Подраздела «Предложение ORDER BY», за исключением того, что выражения в нём всегда принимаются как простые выражения, но не как имя или номер выходного столбца.

Необязательное *предложение_рамки* определяет *рамку окна* для оконных функций, которые зависят от рамки (не все функции таковы). Рамка окна — это набор связанных строк для каждой строки запроса (называемой *текущей строкой*). В качестве *предложения_рамки* может задаваться

```
{ RANGE | ROWS } начало_рамки
{ RANGE | ROWS } BETWEEN начало_рамки AND конец_рамки
```

Здесь *начало_рамки* и *конец_рамки* может задаваться как

```
UNBOUNDED PRECEDING
значение PRECEDING
CURRENT ROW
значение FOLLOWING
UNBOUNDED FOLLOWING
```

Если *конец_рамки* опущен, по умолчанию подразумевается CURRENT ROW. В качестве *начала_рамки* нельзя задать UNBOUNDED FOLLOWING, а в качестве *конца_рамки* — UNBOUNDED PRECEDING, к тому же выбранный вариант *конца_рамки* не может стоять в приведённом списке выше варианта

начала_рамки — например, комбинация RANGE BETWEEN CURRENT ROW AND *значение* PRECEDING не допускается.

По умолчанию рамка образуется предложением RANGE UNBOUNDED PRECEDING, что по сути то же, что RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW; оно устанавливает рамку так, что она включает все строки от начала раздела до последней строки, родственной текущей (строки, которую ORDER BY считает равной текущей), либо все строки раздела, если ORDER BY отсутствует. Вообще UNBOUNDED PRECEDING означает, что рамка начинается с первой строки раздела, а UNBOUNDED FOLLOWING означает, что рамка заканчивается на последней строке раздела (вне зависимости от режима RANGE или ROWS). В режиме ROWS указание CURRENT ROW означает, что рамка начинается или заканчивается текущей строкой; но в режиме RANGE оно означает, что рамка начинается или заканчивается первой или последней строкой, родственной текущей, согласно порядку ORDER BY. Варианты *значение* PRECEDING и *значение* FOLLOWING в настоящее время допускаются только в режиме ROWS. Они показывают, что рамка начинается или заканчивается со сдвигом на заданное количество строк назад или вперёд от текущей. Здесь *значение* должно быть целочисленным выражением, не содержащим никакие переменные, агрегатные или оконные функции. Это значение не может быть отрицательным или равным NULL, но может быть равно 0 (при этом выбирается текущая строка).

Учтите, что варианты ROWS могут выдавать непредсказуемые результаты, если согласно порядку, заданному в ORDER BY, строки сортируются неоднозначно. Варианты RANGE предусмотрены для того, чтобы строки, являющиеся родственными в порядке ORDER BY, обрабатывались одинаково; все родственные строки будут находиться в одной рамке.

Предложение WINDOW применяется для управления поведением *оконных функций*, фигурирующих в запросе, в Подразделе «Список SELECT» или Подразделе «Предложение ORDER BY». Эти функции могут обращаться к элементам WINDOW по именам в своих предложениях OVER. При этом элементы WINDOW не обязательно задействовать в запросе; если они не используются, они просто игнорируются. Оконные функции можно использовать вовсе без элементов WINDOW, так как в вызове оконной функции можно задать определение окна непосредственно в предложении OVER. Однако предложение WINDOW позволяет сократить текст запроса, когда одно и то же определение окна применяется при вызове нескольких оконных функций.

В настоящее время указания FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE нельзя задать вместе с WINDOW.

Оконные функции подробно описываются в Разделе 3.5, Подразделе 4.2.8 и Подразделе 7.2.5.

Список SELECT

Список SELECT (между ключевыми словами SELECT и FROM) содержит выражения, которые формируют выходные строки оператора SELECT. Эти выражения могут обращаться (и обычно обращаются) к столбцам, вычисленным в предложении FROM.

Так же, как в таблице, каждый выходной столбец SELECT имеет имя. В простом предложении SELECT это имя просто помечает столбец при выводе, но когда SELECT представляет собой подзапрос большого запроса, это имя большой запрос видит как имя столбца виртуальной таблицы, созданной подзапросом. Чтобы задать имя для выходного столбца, нужно написать AS *выходное_имя* после выражения столбца. (Слово AS можно опустить, но только если желаемое выходное имя не совпадает с каким-либо ключевым словом Postgres Pro (см. Приложение C). Чтобы не зависеть от появления новых ключевых слов в будущем, рекомендуется всегда писать AS, либо заключать имя в двойные кавычки.) Если имя столбца не задать, Postgres Pro выберет его автоматически. Если выражение столбца представляет собой просто ссылку на столбец, то выбранное таким образом имя будет совпадать с именем столбца. В более сложных случаях может использоваться имя функции или типа, либо в отсутствие других вариантов система может сгенерировать имя вроде ?column?.

По имени выходного столбца можно обратиться к его значению в предложениях ORDER BY и GROUP BY, но не в WHERE или HAVING; в них вместо имени надо записывать всё выражение.

Вместо выражения в выходном списке можно указать *, что будет обозначать все столбцы выбранных строк. Кроме того, можно записать *имя_таблицы*.* как краткое обозначение всех столбцов, получаемых из данной таблицы. В этих случаях нельзя задать новые имена столбцов с помощью AS; именами выходных столбцов будут имена столбцов в таблице.

Согласно стандарту SQL, выражения в выходном списке должны вычисляться до применения DISTINCT, ORDER BY или LIMIT. Это, очевидно, необходимо для DISTINCT, так как иначе не будет ясно, какие значения должны выдаваться как уникальные. Однако во многих случаях выходные выражения удобнее вычислять после ORDER BY и LIMIT; в частности, если в выходном списке содержатся изменчивые или дорогостоящие функции. В этом случае порядок вычисления функций оказывается более интуитивным, а для строк, которые не попадут в результат, не будут производиться вычисления. Postgres Pro фактически будет вычислять выходные выражения после сортировки и ограничения их количества, если эти выражения не фигурируют в DISTINCT, ORDER BY или GROUP BY. (Например, в запросе SELECT f(x) FROM tab ORDER BY 1 функция f(x), несомненно, должна вычисляться перед сортировкой.) Выходные выражения, содержащие функции, возвращающие множества, фактически вычисляются после сортировки и до ограничения количества строк, так что LIMIT будет отбрасывать строки, выдаваемые функцией, возвращающей множество.

> **Примечание**
>
> В Postgres Pro до версии 9.6 никакой порядок вычисления выходных выражений по отношению к сортировке или ограничениям количества не гарантировался; он зависел от формы выбранного плана запроса.

Предложение DISTINCT

Если указано SELECT DISTINCT, все повторяющиеся строки исключаются из результирующего набора (из каждой группы дубликатов остаётся одна строка). SELECT ALL делает противоположное: сохраняет все строки; это поведение по умолчанию.

SELECT DISTINCT ON (*выражение* [, ...]) сохраняет только первую строку из каждого набора строк, для которого данное выражение даёт одинаковые значения. Выражения DISTINCT ON обрабатываются по тем же правилам, что и выражения ORDER BY (см. выше). Заметьте, что «первая строка» каждого набора непредсказуема, если только не применяется предложение ORDER BY, определяющее, какие строки должны быть первыми. Например:

```
SELECT DISTINCT ON (location) location, time, report
    FROM weather_reports
    ORDER BY location, time DESC;
```

возвращает самую последнюю сводку погоды для каждого местоположения. Но если бы мы не добавили ORDER BY, чтобы значения времени убывали, мы бы получили сводки по местоположениям от непредсказуемого времени.

Выражения DISTINCT ON должны соответствовать самым левым выражениям в ORDER BY. Предложение ORDER BY обычно содержит и другие выражения, которые определяют желаемый порядок строк в каждой группе DISTINCT ON.

В настоящее время указания FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE нельзя задать вместе с DISTINCT.

Предложение UNION+

Предложение UNION имеет следующую общую форму:

оператор_SELECT UNION [ALL | DISTINCT] *оператор_SELECT*

Здесь *оператор_SELECT* — это любой подзапрос SELECT без предложений ORDER BY, LIMIT, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE. (ORDER BY и LIMIT можно добавить к вложенному

выражению, если оно заключено в скобки. Без скобок эти предложения будут восприняты как применяемые к результату UNION, а не к выражению в его правой части.)

Оператор UNION вычисляет объединение множеств всех строк, возвращённых заданными запросами SELECT. Строка оказывается в объединении двух наборов результатов, если она присутствует минимум в одном наборе. Два оператора SELECT, представляющие прямые операнды UNION, должны выдавать одинаковое число столбцов, а типы соответствующих столбцов должны быть совместимыми.

Результат UNION не будет содержать повторяющихся строк, если не указан параметр ALL. ALL предотвращает исключение дубликатов. (Таким образом, UNION ALL обычно работает значительно быстрее, чем UNION; поэтому, везде, где возможно, следует указывать ALL.) DISTINCT можно записать явно, чтобы обозначить, что дублирующиеся строки должны удаляться (это поведение по умолчанию).

При использовании в одном запросе SELECT нескольких операторов UNION они вычисляются слева направо, если иной порядок не определяется скобками.

В настоящее время указания FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE нельзя задать ни для результата UNION, ни для любого из подзапросов UNION.

Предложение INTERSECT

Предложение INTERSECT имеет следующую общую форму:

оператор_SELECT INTERSECT [ALL | DISTINCT] *оператор_SELECT*

Здесь *оператор_SELECT* — это любой подзапрос SELECT без предложений ORDER BY, LIMIT, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE.

Оператор INTERSECT вычисляет пересечение множеств всех строк, возвращённых заданными запросами SELECT. Строка оказывается в пересечении двух наборов результатов, если она присутствует в обоих наборах.

Результат INTERSECT не будет содержать повторяющихся строк, если не указан параметр ALL. С параметром ALL строка, повторяющаяся *m* раз в левой таблице и *n* раз в правой, будет выдана в результирующем наборе min(*m*,*n*) раз. DISTINCT можно записать явно, чтобы обозначить, что дублирующиеся строки должны удаляться (это поведение по умолчанию).

При использовании в одном запросе SELECT нескольких операторов INTERSECT они вычисляются слева направо, если иной порядок не диктуется скобками. INTERSECT связывает свои подзапросы сильнее, чем UNION. Другими словами, A UNION B INTERSECT C будет восприниматься как A UNION (B INTERSECT C).

В настоящее время указания FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE нельзя задать ни для результата INTERSECT, ни для любого из подзапросов INTERSECT.

Предложение EXCEPT

Предложение EXCEPT имеет следующую общую форму:

оператор_SELECT EXCEPT [ALL | DISTINCT] *оператор_SELECT*

Здесь *оператор_SELECT* — это любой подзапрос SELECT без предложений ORDER BY, LIMIT, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE.

Оператор EXCEPT вычисляет набор строк, которые присутствуют в результате левого запроса SELECT, но отсутствуют в результате правого.

Результат EXCEPT не будет содержать повторяющихся строк, если не указан параметр ALL. С параметром ALL строка, повторяющаяся *m* раз в левой таблице и *n* раз в правой, будет выдана

в результирующем наборе max(*m*-*n*,0) раз. DISTINCT можно записать явно, чтобы обозначить, что дублирующиеся строки должны удаляться (это поведение по умолчанию).

При использовании в одном запросе SELECT нескольких операторов EXCEPT они вычисляются слева направо, если иной порядок не диктуется скобками. EXCEPT связывает свои подзапросы так же сильно, как UNION.

В настоящее время указания FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE нельзя задать ни для результата EXCEPT, ни для любого из подзапросов EXCEPT.

Предложение ORDER BY

Необязательное предложение ORDER BY имеет следующую общую форму:

```
ORDER BY выражение [ ASC | DESC | USING оператор ] [ NULLS { FIRST | LAST } ] [, ...]
```

Предложение ORDER BY указывает, что строки результата должны сортироваться согласно заданным выражениям. Если две строки дают равные значения для самого левого выражения, проверяется следующее выражение и т. д. Если их значения оказываются равными для всех заданных выражений, строки возвращаются в порядке, определяемом реализацией.

В качестве *выражения* может задаваться имя или порядковый номер выходного столбца (элемента списка SELECT), либо произвольное выражение со значениями входных столбцов.

Порядковым номером в данном случае считается последовательный номер (при нумерации слева направо) позиции выходного столбца. Возможность указать порядковый номер позволяет выполнить сортировку по столбцу, не имеющему уникального имени. В принципе это не абсолютно необходимо, так как выходному столбцу всегда можно присвоить имя, воспользовавшись предложением AS.

В предложении ORDER BY также можно использовать произвольные выражения, в том числе, и со столбцами, отсутствующими в списке результатов SELECT. Таким образом, следующий оператор вполне корректен:

```
SELECT name FROM distributors ORDER BY code;
```

Однако если ORDER BY применяется к результату UNION, INTERSECT или EXCEPT, в нём можно задать только имя или номер выходного столбца, но не выражение.

Если в качестве выражения ORDER BY задано простое имя, которому соответствует и выходной, и входной столбец, то ORDER BY будет воспринимать его как имя выходного столбца. Этот выбор противоположен тому, что делает GROUP BY в такой же ситуации. Такая несогласованность допущена для соответствия стандарту SQL.

Дополнительно после любого выражения в предложении ORDER BY можно добавить ключевое слово ASC (по возрастанию) или DESC (по убыванию). По умолчанию подразумевается ASC. Кроме того, можно задать имя специфического оператора сортировки в предложении USING. Оператор сортировки должен быть членом «меньше» или «больше» некоторого семейства операторов B-дерева. ASC обычно равнозначно USING < и DESC обычно равнозначно USING >. (Хотя создатель нестандартного типа данных может определить по-другому порядок сортировки по умолчанию и поставить ему в соответствие операторы с другими именами.)

Если указано NULLS LAST, значения NULL при сортировке оказываются после значений не NULL; с указанием NULLS FIRST значения NULL оказываются перед значениями не NULL. Если не указано ни то, ни другое, по умолчанию подразумевается NULLS LAST при явно или неявно выбранном порядке ASC, либо NULLS FIRST при порядке DESC (то есть по умолчанию считается, что значения NULL больше значений не NULL). С предложением USING порядок NULL по умолчанию зависит от того, является ли указанный оператор оператором «меньше» или «больше».

Заметьте, что параметры сортировки применяются только к тому выражению, за которым они следуют; в частности, ORDER BY x, y DESC означает не то же самое, что ORDER BY x DESC, y DESC.

Данные символьных строк сортируются согласно правилу сортировки, установленному для сортируемого столбца. При необходимости это правило можно переопределить, добавив предложение COLLATE в *выражение*, например так: ORDER BY mycolumn COLLATE "en_US". За дополнительными сведениями обратитесь к Подразделу 4.2.10 и Разделу 22.2.

Предложение LIMIT

Предложение LIMIT состоит из двух независимых вложенных предложений:

```
LIMIT { число | ALL }
OFFSET начало
```

Здесь *число* определяет максимальное количество строк, которое должно быть выдано, тогда как *начало* определяет, сколько строк нужно пропустить, прежде чем начать выдавать строки. Когда указаны оба значения, сначала строки пропускаются в количестве, заданном значением *начало*, а затем следующие строки выдаются в количестве, не превышающем значения *число*.

Если результатом выражения *число* оказывается NULL, предложение воспринимается как LIMIT ALL, т. е. число строк не ограничивается. Если *начало* принимает значение NULL, предложение воспринимается как OFFSET 0.

SQL:2008 вводит другой синтаксис для получения того же результата, и его так же поддерживает Postgres Pro. Он выглядит так:

```
OFFSET начало { ROW | ROWS }
FETCH { FIRST | NEXT } [ число ] { ROW | ROWS } ONLY
```

В этом синтаксисе значение *начало* или *число* в соответствии со стандартом должно быть буквальной константой, параметром или именем переменной; Postgres Pro позволяет использовать и другие выражения, но их обычно нужно заключать в скобки во избежание неоднозначности. Если *число* опускается в предложении FETCH, оно принимает значение 1. Слова ROW и ROWS, а также FIRST и NEXT являются незначащими и не влияют на поведение этих предложений. Согласно стандарту предложение OFFSET должно идти перед FETCH, если они присутствуют вместе; но Postgres Pro менее строг и допускает любой порядок.

Применяя LIMIT, имеет смысл использовать также предложение ORDER BY, чтобы строки результата выдавались в определённом порядке. Иначе будут возвращаться непредсказуемые подмножества строк запроса — вы можете запросить строки с десятой по двадцатую, но какой порядок вы имеете в виду? Порядок будет неизвестен, если не добавить ORDER BY.

Планировщик запроса учитывает ограничение LIMIT, строя план выполнения запроса, поэтому, вероятнее всего, планы (а значит и порядок строк) будут меняться при разных LIMIT и OFFSET. Таким образом, различные значения LIMIT/OFFSET, выбирающие разные подмножества результатов запроса, *приведут к несогласованности результатов*, если не установить предсказуемую сортировку с помощью ORDER BY. Это не ошибка, а неизбежное следствие того, что SQL не гарантирует вывод результатов запроса в некотором порядке, если порядок не определён явно предложением ORDER BY.

Возможно даже, что при повторном выполнении одного и того же запроса с LIMIT будут получены разные подмножества строк таблицы, если предложение ORDER BY не диктует выбор определённого подмножества. Опять же, это не ошибка; в данном случае детерминированность результата просто не гарантируется.

Предложение блокировки

Предложения блокировки включают в себя FOR UPDATE, FOR NO KEY UPDATE, FOR SHARE и FOR KEY SHARE; они влияют на то, как SELECT блокирует строки, получаемые из таблицы.

Предложение блокировки имеет следующую общую форму:

```
FOR вариант_блокировки [ OF имя_таблицы [, ...] ] [ NOWAIT | SKIP LOCKED ]
```

Здесь *вариант_блокировки* может быть следующим:

```
UPDATE
NO KEY UPDATE
SHARE
KEY SHARE
```

Подробнее о каждом режиме блокировки на уровне строк можно узнать в Подразделе 13.3.2.

Чтобы операция не ждала завершения других транзакций, к блокировке можно добавить указание NOWAIT или SKIP LOCKED. С NOWAIT оператор выдаёт ошибку, а не ждёт, если выбранную строку нельзя заблокировать немедленно. С указанием SKIP LOCKED выбранные строки, которые нельзя заблокировать немедленно, пропускаются. При этом формируется несогласованное представление данных, так что этот вариант не подходит для общего применения, но может использоваться для исключения блокировок при обращении множества потребителей к таблице типа очереди. Заметьте, что указания NOWAIT и SKIP LOCKED применяются только к блокировкам на уровне строк — необходимая блокировка ROW SHARE уровня таблицы запрашивается обычным способом (см. Главу 13). Если требуется запросить блокировку уровня таблицы без ожидания, можно сначала выполнить LOCK с указанием NOWAIT.

Если в предложении блокировки указаны определённые таблицы, блокироваться будут только строки, получаемые из этих таблиц; другие таблицы, задействованные в SELECT, будут прочитаны как обычно. Предложение блокировки без списка таблиц затрагивает все таблицы, задействованные в этом операторе. Если предложение блокировки применяется к представлению или подзапросу, оно затрагивает все таблицы, которые используются в представлении или подзапросе. Однако эти предложения не применяются к запросам WITH, к которым обращается основной запрос. Если требуется установить блокировку строк в запросе WITH, предложение блокировки нужно указать непосредственно в этом запросе WITH.

В случае необходимости задать для разных таблиц разное поведение блокировки, в запрос можно добавить несколько предложений. Если при этом одна и та же таблица упоминается (или неявно затрагивается) в нескольких предложениях блокировки, блокировка устанавливается так, как если бы было указано только одно, самое сильное из них. Подобным образом, если в одном из предложений указано NOWAIT, для этой таблицы блокировка будет запрашиваться без ожидания. В противном случае она будет обработана в режиме SKIP LOCKED, если он выбран в любом из затрагивающих её предложений.

Предложения блокировки не могут применяться в контекстах, где возвращаемые строки нельзя чётко связать с отдельными строками таблицы; например, блокировка неприменима при агрегировании.

Когда предложение блокировки находится на верхнем уровне запроса SELECT, блокируются именно те строки, которые возвращаются запросом; в случае с запросом объединения, блокировке подлежат строки, из которых составляются возвращаемые строки объединения. В дополнение к этому, заблокированы будут строки, удовлетворяющие условиям запроса на момент создания снимка запроса, хотя они не будут возвращены, если с момента снимка они изменятся и перестанут удовлетворять условиям. Если применяется LIMIT, блокировка прекращается, как только будет получено достаточное количество строк для удовлетворения лимита (но заметьте, что строки, пропускаемые указанием OFFSET, будут блокироваться). Подобным образом, если предложение блокировки применяется в запросе курсора, блокироваться будут только строки, фактически полученные или пройденные курсором.

Когда предложение блокировки находится в подзапросе SELECT, блокировке подлежат те строки, которые будет получены внешним запросом от подзапроса. Таких строк может оказаться меньше, чем можно было бы предположить, проанализировав только сам подзапрос, так как условия из внешнего запроса могут способствовать оптимизации выполнения подзапроса. Например, запрос

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss WHERE col1 = 5;
```

заблокирует только строки, в которых col1 = 5, при том, что в такой записи условие не относится к подзапросу.

Предыдущие версии не могли сохранить блокировку, которая была повышена последующей точкой сохранения. Например, этот код:

```
BEGIN;
SELECT * FROM mytable WHERE key = 1 FOR UPDATE;
SAVEPOINT s;
UPDATE mytable SET ... WHERE key = 1;
ROLLBACK TO s;
```

не мог сохранить блокировку FOR UPDATE после ROLLBACK TO. Это было исправлено в версии 9.3.

> **Внимание**
>
> Возможно, что команда SELECT, работающая на уровне изоляции READ COMMITTED и применяющая предложение ORDER BY вместе с блокировкой, будет возвращать строки не по порядку. Это связано с тем, что ORDER BY выполняется в первую очередь. Эта команда отсортирует результат, но затем может быть заблокирована, пытаясь получить блокировку одной или нескольких строк. К моменту, когда блокировка SELECT будет снята, некоторые из сортируемых столбцов могут уже измениться, в результате чего их порядок может быть нарушен (хотя они были упорядочены для исходных значений). При необходимости обойти эту проблему, можно поместить FOR UPDATE/SHARE в подзапрос, например так:
>
> ```
> SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss ORDER BY column1;
> ```
>
> Заметьте, что в результате это приведёт к блокированию всех строк в mytable, тогда как указание FOR UPDATE на верхнем уровне могло бы заблокировать только фактически возвращаемые строки. Это может значительно повлиять на производительность, особенно в сочетании ORDER BY с LIMIT или другими ограничениями. Таким образом, этот приём рекомендуется, только если ожидается параллельное изменение сортируемых столбцов, а результат должен быть строго отсортирован.
>
> На уровнях изоляции REPEATABLE READ и SERIALIZABLE это приведёт к ошибке сериализации (с SQLSTATE '40001'), так что на этих уровнях получить строки не по порядку невозможно.

Команда TABLE

Команда

```
TABLE имя
```

равнозначна

```
SELECT * FROM имя
```

Её можно применять в качестве команды верхнего уровня или как более краткую запись внутри сложных запросов. С командой TABLE могут использоваться только предложения WITH, UNION, INTERSECT, EXCEPT, ORDER BY, LIMIT, OFFSET, FETCH и предложения блокировки FOR; предложение WHERE и какие-либо формы агрегирования не поддерживаются.

Примеры

Соединение таблицы films с таблицей distributors:

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
    FROM distributors d, films f
    WHERE f.did = d.did

       title       | did |     name     | date_prod  |   kind
-------------------+-----+--------------+------------+----------
 The Third Man     | 101 | British Lion | 1949-12-23 | Drama
 The African Queen | 101 | British Lion | 1951-08-11 | Romantic
 ...
```

Суммирование значений столбца `len` (продолжительность) для всех фильмов и группирование результатов по столбцу `kind` (типу фильма):

```
SELECT kind, sum(len) AS total FROM films GROUP BY kind;
```

```
   kind   | total
----------+-------
 Action   | 07:34
 Comedy   | 02:58
 Drama    | 14:28
 Musical  | 06:42
 Romantic | 04:38
```

Суммирование значений столбца `len` для всех фильмов, группирование результатов по столбцу `kind` и вывод только тех групп, общая продолжительность которых меньше 5 часов:

```
SELECT kind, sum(len) AS total
    FROM films
    GROUP BY kind
    HAVING sum(len) < interval '5 hours';
```

```
   kind   | total
----------+-------
 Comedy   | 02:58
 Romantic | 04:38
```

Следующие два запроса демонстрируют равнозначные способы сортировки результатов по содержимому второго столбца (`name`):

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;
```

```
 did |       name
-----+------------------
 109 | 20th Century Fox
 110 | Bavaria Atelier
 101 | British Lion
 107 | Columbia
 102 | Jean Luc Godard
 113 | Luso films
 104 | Mosfilm
 103 | Paramount
 106 | Toho
 105 | United Artists
 111 | Walt Disney
 112 | Warner Bros.
 108 | Westward
```

Следующий пример показывает объединение таблиц `distributors` и `actors`, ограниченное именами, начинающимися с буквы W в каждой таблице. Интерес представляют только неповторяющиеся строки, поэтому ключевое слово `ALL` опущено.

```
distributors:              actors:
 did |     name             id |     name
-----+--------------       ----+----------------
 108 | Westward               1 | Woody Allen
 111 | Walt Disney            2 | Warren Beatty
 112 | Warner Bros.           3 | Walter Matthau
 ...                        ...
```

```
SELECT distributors.name
    FROM distributors
    WHERE distributors.name LIKE 'W%'
UNION
SELECT actors.name
    FROM actors
    WHERE actors.name LIKE 'W%';

      name
----------------
 Walt Disney
 Walter Matthau
 Warner Bros.
 Warren Beatty
 Westward
 Woody Allen
```

Этот пример показывает, как использовать функцию в предложении `FROM`, со списком определений столбцов и без него:

```
CREATE FUNCTION distributors(int) RETURNS SETOF distributors AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributors(111);
 did |    name
-----+-------------
 111 | Walt Disney

CREATE FUNCTION distributors_2(int) RETURNS SETOF record AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributors_2(111) AS (f1 int, f2 text);
 f1  |     f2
-----+-------------
 111 | Walt Disney
```

Пример функции с добавленным столбцом нумерации:

```
SELECT * FROM unnest(ARRAY['a','b','c','d','e','f']) WITH ORDINALITY;
 unnest | ordinality
--------+----------
 a      |        1
 b      |        2
 c      |        3
 d      |        4
 e      |        5
 f      |        6
(6 rows)
```

Этот пример показывает, как использовать простое предложение `WITH`:

```
WITH t AS (
    SELECT random() as x FROM generate_series(1, 3)
  )
SELECT * FROM t
UNION ALL
SELECT * FROM t
```

```
         x
--------------------
  0.534150459803641
  0.520092216785997
 0.0735620250925422
  0.534150459803641
  0.520092216785997
 0.0735620250925422
```

Заметьте, что запрос `WITH` выполняется всего один раз, поэтому мы получаем два одинаковых набора по три случайных значения.

В этом примере `WITH RECURSIVE` применяется для поиска всех подчинённых Мери (непосредственных или косвенных) и вывода их уровня косвенности в таблице с информацией только о непосредственных подчинённых:

```
WITH RECURSIVE employee_recursive(distance, employee_name, manager_name) AS (
    SELECT 1, employee_name, manager_name
    FROM employee
    WHERE manager_name = 'Mary'
  UNION ALL
    SELECT er.distance + 1, e.employee_name, e.manager_name
    FROM employee_recursive er, employee e
    WHERE er.employee_name = e.manager_name
  )
SELECT distance, employee_name FROM employee_recursive;
```

Заметьте, что это типичная форма рекурсивных запросов: начальное условие, последующий `UNION`, а затем рекурсивная часть запроса. Убедитесь в том, что рекурсивная часть запроса в конце концов перестанет возвращать строки, иначе запрос окажется в бесконечном цикле. (За другими примерами обратитесь к Разделу 7.8.)

В этом примере используется `LATERAL` для применения функции `get_product_names()`, возвращающей множество, для каждой строки таблицы `manufacturers`:

```
SELECT m.name AS mname, pname
FROM manufacturers m, LATERAL get_product_names(m.id) pname;
```

Производители, с которыми в данный момент не связаны никакие продукты, не попадут в результат, так как это внутреннее соединение. Если бы мы захотели включить названия и этих производителей, мы могли бы сделать так:

```
SELECT m.name AS mname, pname
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true;
```

Совместимость

Разумеется, оператор `SELECT` совместим со стандартом SQL. Однако не все описанные в стандарте возможности реализованы, а некоторые, наоборот, являются расширениями.

Необязательное предложение `FROM`

Postgres Pro разрешает опустить предложение `FROM`. Это позволяет очень легко вычислять результаты простых выражений:

```
SELECT 2+2;

 ?column?
----------
        4
```

Некоторые другие базы данных SQL не допускают этого, требуя задействовать в `SELECT` фиктивную таблицу с одной строкой.

Заметьте, что если предложение FROM не указано, запрос не может обращаться ни к каким таблицам базы данных. Например, следующий запрос недопустим:

```
SELECT distributors.* WHERE distributors.name = 'Westward';
```

До версии 8.1 PostgreSQL мог принимать запросы такого вида, неявно добавляя каждую таблицу, задействованную в запросе, в предложение FROM этого запроса. Теперь это не допускается.

Пустые списки SELECT

Список выходных выражений после SELECT может быть пустым, что в результате даст таблицу без столбцов. Стандарт SQL не считает такой синтаксис допустимым, но Postgres Pro допускает его, так как это согласуется с возможностью иметь таблицы с нулём столбцов. Однако когда используется DISTINCT, пустой список не допускается.

Необязательное ключевое слово AS

В стандарте SQL необязательное ключевое слово AS можно опустить перед именем выходного столбца, если это имя является допустимым именем столбца (то есть не совпадает с каким-либо зарезервированным ключевым словом). Postgres Pro несколько более строг: AS требуется, если имя столбца совпадает с любым ключевым словом, зарезервированным или нет. Тем не менее рекомендуется использовать AS или заключать имена выходных столбцов в кавычки, во избежание конфликтов, возможных при появлении в будущем новых ключевых слов.

В списке FROM и стандарт, и Postgres Pro позволяют опускать AS перед псевдонимом, который является незарезервированным ключевым словом. Но для имён выходных столбцов это не подходит из-за синтаксической неоднозначности.

ONLY и наследование

Стандарт SQL требует заключать в скобки имя таблицы после ONLY, например SELECT * FROM ONLY (tab1), ONLY (tab2) WHERE .... Postgres Pro считает эти скобки необязательными.

Postgres Pro позволяет добавлять в конце *, чтобы явно обозначить, что дочерние таблицы включаются в рассмотрение, в отличие от поведения с ONLY. Стандарт не позволяет этого.

(Эти соображения в равной степени касаются всех SQL-команд, поддерживающих параметр ONLY.)

Ограничения предложения TABLESAMPLE

Предложение TABLESAMPLE в настоящий момент принимается только для обычных таблиц и материализованных представлений. Однако согласно стандарту SQL оно должно применяться к любым элементам списка FROM.

Вызовы функций в предложении FROM

Postgres Pro позволяет записать вызов функции непосредственно в виде элемента списка FROM. В стандарте SQL такой вызов функции требуется помещать во вложенный SELECT; то есть, запись FROM *функция*(...) *псевдоним* примерно равнозначна записи FROM LATERAL (SELECT *функция*(...)) *псевдоним*. Заметьте, что указание LATERAL считается неявным; это связано с тем, что стандарт требует поведения LATERAL для элемента UNNEST() в предложении FROM. Postgres Pro обрабатывает UNNEST() так же, как и другие функции, возвращающие множества.

Пространства имён в GROUP BY и ORDER BY

В стандарте SQL-92 предложение ORDER BY может содержать ссылки только на выходные столбцы по именам или номерам, тогда как GROUP BY может содержать выражения с именами только входных столбцов. Postgres Pro расширяет оба эти предложения, позволяя также применять другие варианты (но если возникает неоднозначность, он разрешает её согласно стандарту). Postgres Pro также позволяет задавать произвольные выражения в обоих предложениях. Заметьте, что имена, фигурирующие в выражениях, всегда будут восприниматься как имена входных, а не выходных столбцов.

В SQL:1999 и более поздних стандартах введено несколько другое определение, которое не полностью совместимо с SQL-92. Однако в большинстве случаев Postgres Pro будет интерпретировать выражение ORDER BY или GROUP BY так, как требует SQL:1999.

Функциональные зависимости

Postgres Pro распознаёт функциональную зависимость (что позволяет опускать столбцы в GROUP BY), только когда первичный ключ таблицы присутствует в списке GROUP BY. В стандарте SQL оговариваются дополнительные условия, которые следует учитывать.

Ограничения предложения WINDOW

Стандарт SQL предоставляет дополнительные возможности для указания *предложения_рамки* окна. Postgres Pro в настоящее время поддерживает только варианты, описанные выше.

LIMIT и OFFSET

Предложения LIMIT и OFFSET относятся к специфическим особенностям Postgres Pro и поддерживаются также в MySQL. В стандарте SQL:2008 для той же цели вводятся предложения OFFSET ... FETCH {FIRST|NEXT} ..., рассмотренные ранее в Подразделе «Предложение LIMIT». Этот синтаксис также используется в IBM DB2. (Приложения, написанные для Oracle, часто применяют обходной способ и получают эффект этих предложений, задействуя автоматически генерируемый столбец rownum, который отсутствует в Postgres Pro.)

FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE, FOR KEY SHARE

Хотя указание FOR UPDATE есть в стандарте SQL, стандарт позволяет использовать его только в предложении DECLARE CURSOR. Postgres Pro допускает его использование в любом запросе SELECT, а также в подзапросах SELECT, но это является расширением. Варианты FOR NO KEY UPDATE, FOR SHARE и FOR KEY SHARE, а также указания NOWAIT и SKIP LOCKED в стандарте отсутствуют.

Изменение данных в WITH

Postgres Pro разрешает использовать INSERT, UPDATE и DELETE в качестве запросов WITH. Стандарт SQL этого не предусматривает.

Нестандартные предложения

DISTINCT ON (...) — расширение стандарта SQL.

ROWS FROM(...) — расширение стандарта SQL.

SELECT INTO

SELECT INTO — создать таблицу из результатов запроса

Синтаксис

```
[ WITH [ RECURSIVE ] запрос_WITH [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( выражение [, ...] ) ] ]
    * | выражение [ [ AS ] имя_результата ] [, ...]
    INTO [ TEMPORARY | TEMP | UNLOGGED ] [ TABLE ] новая_таблица
    [ FROM элемент_FROM [, ...] ]
    [ WHERE условие ]
    [ GROUP BY выражение [, ...] ]
    [ HAVING условие ]
    [ WINDOW имя_окна AS ( определение_окна ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] выборка ]
    [ ORDER BY выражение [ ASC | DESC | USING оператор ] [ NULLS { FIRST | LAST } ]
 [, ...] ]
    [ LIMIT { число | ALL } ]
    [ OFFSET начало [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ число ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | SHARE } [ OF имя_таблицы [, ...] ] [ NOWAIT ] [...] ]
```

Описание

`SELECT INTO` создаёт новую таблицу и заполняет её данными, полученными из запроса. Данные не передаются клиенту, как с обычной командой `SELECT`. Столбцы новой таблицы получают имена и типы данных, связанные с выходными столбцами `SELECT`.

Параметры

`TEMPORARY` или `TEMP`

Если указано, создаваемая таблица будет временной. За подробностями обратитесь к CREATE TABLE.

`UNLOGGED`

Если указано, создаваемая таблица будет нежурналируемой. За подробностями обратитесь к CREATE TABLE.

`новая_таблица`

Имя создаваемой таблицы (возможно, дополненное схемой).

Все другие параметры подробно описываются в SELECT.

Замечания

Команда `SELECT INTO` действует подобно CREATE TABLE AS, но рекомендуется использовать `CREATE TABLE AS`, так как `SELECT INTO` не поддерживается в ECPG и PL/pgSQL, вследствие того, что они воспринимают предложение `INTO` по-своему. К тому же, `CREATE TABLE AS` предоставляет больший набор возможностей, чем `SELECT INTO`.

Чтобы добавить столбец OID в таблицу, создаваемую командой `SELECT INTO`, необходимо установить конфигурационную переменную default_with_oids. С другой стороны, можно использовать `CREATE TABLE AS` с предложением `WITH OIDS`.

Примеры

Создание таблицы `films_recent`, содержащей только последние записи из таблицы `films`:

```
SELECT * INTO films_recent FROM films WHERE date_prod >= '2002-01-01';
```

Совместимость

В стандарте SQL команда `SELECT INTO` применяется для передачи скалярных значений клиентской программе, но не для создания новой таблицы. Именно это применение имеет место в ECPG (см. Главу 33) и в PL/pgSQL (см. Главу 40). В Postgres Pro команда `SELECT INTO` связана с созданием таблицы по историческим причинам. В новом коде для этих целей лучше использовать `CREATE TABLE AS`.

См. также

CREATE TABLE AS

SET

SET — изменить параметр времени выполнения

Синтаксис

```
SET [ SESSION | LOCAL ] параметр_конфигурации { TO | = } { значение | 'значение' |
 DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { значение | 'значение' | LOCAL | DEFAULT }
```

Описание

Команда SET изменяет конфигурационные параметры времени выполнения. С помощью SET можно «на лету» изменить многие из параметров, перечисленных в Главе 18. (Но для изменения некоторых могут потребоваться права суперпользователя, а другие нельзя изменять после запуска сервера или сеанса.) SET влияет на значение параметра только в рамках текущего сеанса.

Если команда SET (или равнозначная SET SESSION) выполняется внутри транзакции, которая затем прерывается, эффект команды SET пропадает, когда транзакция откатывается. Если же окружающая транзакция фиксируется, этот эффект сохраняется до конца сеанса, если его не переопределит другая команда SET.

Действие SET LOCAL продолжается только до конца текущей транзакции, независимо от того, фиксируется она или нет. Особый случай представляет использование SET с последующей SET LOCAL в одной транзакции: значение, заданное SET LOCAL, будет сохраняться до конца транзакции, но после этого (если транзакция фиксируется) восстановится значение, заданное командой SET.

Действия SET или SET LOCAL также отменяются при откате к точке сохранения, установленной до выполнения этих команд.

Если SET LOCAL применяется в функции, параметр SET для которой устанавливает значение той же переменной (см. CREATE FUNCTION), действие команды SET LOCAL прекращается при выходе из функции; то есть, в любом случае восстанавливается значение, существовавшее при вызове функции. Это позволяет использовать SET LOCAL для динамических и неоднократных изменений параметра в рамках функции, и при этом иметь удобную возможность использовать параметр SET для сохранения и восстановления значения, полученного извне. Однако обычная команда SET переопределяет любой параметр SET окружающей функции; её действие сохраняется, если не происходит откат транзакции.

> **Примечание**
>
> В PostgreSQL с версии 8.0 до 8.2 действие SET LOCAL могло отменяться при освобождении ранее установленной точки сохранения или при успешном выходе из блока исключения PL/pgSQL. Затем это поведение было признано неинтуитивным, и было изменено.

Параметры

SESSION

Указывает, что команда действует в рамках текущего сеанса. (Это поведение по умолчанию, если не указано ни SESSION, ни LOCAL.)

LOCAL

Указывает, что команда действует только в рамках текущей транзакции. После COMMIT или ROLLBACK в силу вновь вступает значение, определённое на уровне сеанса. При выполнении

такой команды вне блока транзакции выдаётся предупреждение и больше ничего не происходит.

параметр_конфигурации

Имя устанавливаемого параметра времени выполнения. Доступные параметры описаны в Главе 18 и ниже.

значение

Новое значение параметра. Параметры могут задаваться в виде строковых констант, идентификаторов, чисел или списков перечисленных типов через запятую, в зависимости от конкретного параметра. Указание `DEFAULT` в данном контексте позволяет сбросить параметр к значению по умолчанию (то есть, к тому значению, которое он имел бы, если в текущем сеансе не выполнялись бы команды `SET`).

Помимо конфигурационных параметров, описанных в Главе 18, есть ещё несколько параметров, которые можно изменить только командой `SET` или которые имеют особый синтаксис:

`SCHEMA`

`SET SCHEMA '`*значение*`'` — альтернативное написание команды `SET search_path TO` *значение*. Такой синтаксис позволяет указать только одну схему.

`NAMES`

`SET NAMES` *значение* — альтернативное написание команды `SET client_encoding TO` *значение*.

`SEED`

Устанавливает внутреннее начальное значение для генератора случайных чисел (функции `random`). В качестве значения допускаются числа с плавающей точкой от -1 до 1, для внутреннего применения они затем умножаются на 2^{31}-1.

Это начальное значение также можно установить, вызвав функцию `setseed`:

```
SELECT setseed(значение);
```

`TIME ZONE`

`SET TIME ZONE '`*значение*`'` — альтернативное написание команды `SET timezone TO '`*значение*`'`. Синтаксис `SET TIME ZONE` позволяет указывать часовой пояс в специальном формате. Например, допускаются следующие значения:

`'PST8PDT'`

Часовой пояс Беркли, штат Калифорния.

`'Europe/Rome'`

Часовой пояс Италии.

`-7`

Часовой пояс, сдвинутый от UTC на 7 часов к западу (равнозначен PDT). Положительные значения означают сдвиг от UTC к востоку.

`INTERVAL '-08:00' HOUR TO MINUTE`

Часовой пояс, сдвинутый от UTC на 8 часов к западу (равнозначен PST).

`LOCAL`
`DEFAULT`

Устанавливает в качестве часового пояса местный часовой пояс (то есть, значение серверного параметра `timezone` по умолчанию).

Значения часового пояса, заданные в виде чисел или интервалов, внутри переводятся в формат часовых поясов POSIX. Например, после `SET TIME ZONE -7`, команда `SHOW TIME ZONE` покажет `<-07>+07`.

Аббревиатуры часовых поясов команда `SET` не поддерживает; узнать о часовых поясах подробнее можно в Подразделе 8.5.3.

Замечания

Также изменить значение параметра можно с помощью функции `set_config`; см. Раздел 9.26. Кроме того, выполнив UPDATE в системном представлении `pg_settings`, можно произвести то же действие, что выполняет `SET`.

Примеры

Установка пути поиска схем:

```
SET search_path TO my_schema, public;
```

Установка традиционного стиля даты POSTGRES с форматом ввода «день, месяц, год»:

```
SET datestyle TO postgres, dmy;
```

Установка часового пояса для Беркли, штат Калифорния:

```
SET TIME ZONE 'PST8PDT';
```

Установка часового пояса Италии:

```
SET TIME ZONE 'Europe/Rome';
```

Совместимость

`SET TIME ZONE` расширяет синтаксис, определённый в стандарте SQL. Стандарт допускает только числовые смещения часовых поясов, тогда как Postgres Pro позволяет задавать часовой пояс более гибко. Все другие функции `SET` являются расширениями Postgres Pro.

См. также

RESET, SHOW

SET CONSTRAINTS

SET CONSTRAINTS — установить время проверки ограничений для текущей транзакции

Синтаксис

```
SET CONSTRAINTS { ALL | имя [, ...] } { DEFERRED | IMMEDIATE }
```

Описание

`SET CONSTRAINTS` определяет, когда будут проверяться ограничения в текущей транзакции. Ограничения `IMMEDIATE` проверяются в конце каждого оператора, а ограничения `DEFERRED` откладываются до фиксации транзакции. Режим `IMMEDIATE` или `DEFERRED` задаётся для каждого ограничения независимо.

При создании ограничение получает одну из следующих характеристик: `DEFERRABLE INITIALLY DEFERRED` (откладываемое, изначально отложенное), `DEFERRABLE INITIALLY IMMEDIATE` (откладываемое, изначально немедленное) или `NOT DEFERRABLE` (неоткладываемое). Третий вариант всегда подразумевает `IMMEDIATE` и на него команда `SET CONSTRAINTS` не влияет. Первые два варианта запускаются в каждой транзакции в указанном режиме, но их поведение можно изменить в рамках транзакции командой `SET CONSTRAINTS`.

`SET CONSTRAINTS` со списком имён ограничений меняет режим только этих ограничений (все они должны быть откладываемыми). Имя любого ограничения можно дополнить схемой. Если имя схемы не указано, в поисках первого подходящего имени будет просматриваться текущий путь поиска схем. `SET CONSTRAINTS ALL` меняет режим всех откладываемых ограничений.

Когда `SET CONSTRAINTS` меняет режим ограничения с `DEFERRED` на `IMMEDIATE`, новый режим начинает действовать в обратную сторону: все изменения данных, ожидающие проверки в конце транзакции, вместо этого проверяются в момент выполнения команды `SET CONSTRAINTS`. Если какое-либо ограничение нарушается, при выполнении `SET CONSTRAINTS` происходит ошибка (и режим проверки не меняется). Таким образом, с помощью `SET CONSTRAINTS` можно принудительно проверить ограничения в определённом месте транзакции.

В настоящее время это распространяется только на ограничения `UNIQUE`, `PRIMARY KEY`, `REFERENCES` (внешний ключ) и `EXCLUDE`. Ограничения `NOT NULL` и `CHECK` всегда проверяются немедленно в момент добавления или изменения строки (*не* в конце оператора). Ограничения уникальности и ограничения-исключения, объявленные без указания `DEFERRABLE`, так же проверяются немедленно.

Срабатывание триггеров, объявленных как «триггеры ограничений» так же зависит от этой команды — они срабатывают в момент, когда должно проверяться соответствующее ограничение.

Замечания

Так как Postgres Pro не требует, чтобы имена ограничений были уникальны в схеме (достаточно уникальности в таблице), возможно, что для заданного имени найдётся несколько соответствующих ограничений. В этом случае `SET CONSTRAINTS` подействует на все эти ограничения. Для имён без указания схемы, её действие будет распространяться только на ограничение(я), найденное в первой из схем; другие схемы просматриваться не будут.

Эта команда меняет поведение ограничений только в текущей транзакции. При выполнении этой команды вне блока транзакции выдаётся предупреждение и больше ничего не происходит.

Совместимость

Эта команда реализует поведение, описанное в стандарте SQL, с одним исключением — в Postgres Pro она не влияет на проверку ограничений `NOT NULL` и `CHECK`. Кроме того, Postgres Pro проверяет

неоткладываемые ограничения уникальности немедленно, а не в конце оператора, как предлагает стандарт.

SET ROLE

SET ROLE — установить идентификатор текущего пользователя в рамках сеанса

Синтаксис

```
SET [ SESSION | LOCAL ] ROLE имя_роли
SET [ SESSION | LOCAL ] ROLE NONE
RESET ROLE
```

Описание

Эта команда меняет идентификатор текущего пользователя в активном сеансе SQL на *имя_роли*. Имя роли может быть записано в виде идентификатора или строковой константы. После SET ROLE, права доступа для команд SQL проверяются так, как если бы сеанс изначально был установлен с этим именем роли.

Указывая определённое *имя_роли*, текущий пользователь должен являться членом этой роли. (Если пользователь сеанса является суперпользователем, он может выбрать любую роль.)

Указания SESSION и LOCAL действуют на эту команду так же, как и на обычную команду SET.

SET ROLE NONE устанавливает в качестве идентификатора текущего пользователя идентификатор текущего пользователя сеанса, выдаваемый функцией session_user. RESET ROLE устанавливает в качестве идентификатора текущего пользователя значение, заданное во время подключения параметрами командной строки либо командой ALTER ROLE или ALTER DATABASE. Если же такое значение не задано, в качестве идентификатора текущего пользователя так же устанавливается идентификатор текущего пользователя сеанса. Эти формы могут выполняться любыми пользователями.

Замечания

С помощью этой команды можно как добавить права, так и ограничить их. Если роль пользователя сеанса имеет атрибут INHERIT, она автоматически получает права всех ролей, на которые может переключиться (с помощью SET ROLE); в этом случае SET ROLE убирает все права, назначенные непосредственно пользователю сеанса и другим ролям, в которые он включён, и оставляет только права, назначенные указанной роли. Если же роль пользователя сеанса имеет атрибут NOINHERIT, SET ROLE убирает права, назначенные непосредственно пользователю сеанса, и вместо них назначает права, которые имеет указанная роль.

В частности, когда суперпользователь переключается на роль не суперпользователя (используя SET ROLE), он теряет свои права суперпользователя.

SET ROLE оказывает действие, сравнимое с SET SESSION AUTHORIZATION, но проверка прав выполняется по-другому. Также SET SESSION AUTHORIZATION определяет, какие роли разрешены для последующей SET ROLE, тогда как при смене ролей командой SET ROLE набор ролей, допустимых для последующей команды SET ROLE, не меняется.

SET ROLE не обрабатывает сеансовые переменные, указанные в свойствах роли (ALTER ROLE); они устанавливаются только при подключении.

SET ROLE нельзя использовать внутри функции с характеристикой SECURITY DEFINER.

Примеры

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
```

```
--------------+--------------
 peter        | peter

SET ROLE 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | paul
```

Совместимость

Postgres Pro принимает идентификаторы ("*имя_роли*"), тогда как стандарт SQL требует, чтобы имя роли записывалось в виде строковой константы. Стандарт не позволяет выполнять эту команду в транзакции; в Postgres Pro такого ограничения нет, так как для него нет причины. Указания `SESSION` и `LOCAL` относятся к расширениям Postgres Pro, так же, как и синтаксис `RESET`.

См. также

SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION — установить идентификатор пользователя сеанса и идентификатор текущего пользователя в рамках сеанса

Синтаксис

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION имя_пользователя
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

Описание

Эта команда меняет идентификатор пользователя сеанса и идентификатор текущего пользователя в рамках активного сеанса SQL на *имя_пользователя*. Имя пользователя может быть записано в виде идентификатора или строковой константы. С помощью этой команды, можно, например, временно переключиться на непривилегированного пользователя, сохранив возможность затем стать суперпользователем.

Идентификатором пользователя сеанса изначально принимается имя пользователя, введённое клиентом (возможно, прошедшее проверку подлинности). Идентификатор текущего пользователя обычно совпадает с идентификатором пользователя сеанса, но может временно меняться в функциях с характеристикой `SECURITY DEFINER` и подобными механизмами; также его можно изменить командой SET ROLE. Идентификатор текущего пользователя принимается во внимание при проверке разрешений.

Идентификатор пользователя сеанса можно изменить, только если начальный пользователь сеанса (*аутентифицированный пользователь*) имеет права суперпользователя. В противном случае эта команда разрешается, только если в ней указывается имя аутентифицированного пользователя.

Указания `SESSION` и `LOCAL` действуют на эту команду так же, как и на обычную команду SET.

Формы `DEFAULT` и `RESET` сбрасывают идентификаторы текущего пользователя и пользователя сеанса, так что текущим становится пользователь, изначально проходивший проверку подлинности. Эти формы могут выполняться любым пользователем.

Замечания

`SET SESSION AUTHORIZATION` нельзя использовать в функции с характеристикой `SECURITY DEFINER`.

Примеры

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 paul         | paul
```

Совместимость

Стандарт SQL позволяет вместо строковой константы *имя_пользователя* указывать некоторые другие выражения, но на практике это не очень полезно. Postgres Pro допускает синтаксис идентификаторов ("*имя_пользователя*"), а стандарт SQL — нет. Стандарт не позволяет выполнять эту команду в транзакции; в Postgres Pro такого ограничения нет, так как для него нет причины. Указания `SESSION` и `LOCAL` относятся к расширениям Postgres Pro, так же, как и синтаксис `RESET`.

Набор прав, требуемых для выполнения этой команды, согласно стандарту, определяется реализацией.

См. также

SET ROLE

SET TRANSACTION

SET TRANSACTION — установить характеристики текущей транзакции

Синтаксис

```
SET TRANSACTION режим_транзакции [, ...]
SET TRANSACTION SNAPSHOT id_снимка
SET SESSION CHARACTERISTICS AS TRANSACTION режим_транзакции [, ...]

Где режим_транзакции может быть следующим:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
 UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

Описание

Команда `SET TRANSACTION` устанавливает характеристики текущей транзакции. На последующие транзакции она не влияет. `SET SESSION CHARACTERISTICS` устанавливает характеристики транзакции по умолчанию для последующих транзакций в рамках сеанса. Заданные по умолчанию характеристики затем можно переопределить для отдельных транзакций командой `SET TRANSACTION`.

К характеристикам транзакции относится уровень изоляции транзакции, режим доступа транзакции (чтение/запись или только чтение) и допустимость её откладывания. В дополнение к ним можно выбрать снимок, но только для текущей транзакции, не для сеанса по умолчанию.

Уровень изоляции транзакции определяет, какие данные может видеть транзакция, когда параллельно с ней выполняются другие транзакции:

`READ COMMITTED`

Оператор видит только те строки, которые были зафиксированы до начала его выполнения. Этот уровень устанавливается по умолчанию.

`REPEATABLE READ`

Все операторы текущей транзакции видят только те строки, которые были зафиксированы перед первым запросом на выборку или изменение данных, выполненным в этой транзакции.

`SERIALIZABLE`

Все операторы текущей транзакции видят только те строки, которые были зафиксированы перед первым запросом на выборку или изменение данных, выполненным в этой транзакции. Если наложение операций чтения и записи параллельных сериализуемых транзакций может привести к ситуации, невозможной при последовательном их выполнении (когда одна транзакция выполняется за другой), произойдёт откат одной из транзакций с ошибкой `serialization_failure` (сбой сериализации).

В стандарте SQL определён ещё один уровень, `READ UNCOMMITTED`. В Postgres Pro уровень `READ UNCOMMITTED` обрабатывается как `READ COMMITTED`.

Уровень изоляции транзакции нельзя изменить после выполнения первого запроса на выборку или изменение данных (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH` или `COPY`) в текущей транзакции. За дополнительными сведениями об изоляции транзакций и управлении параллельным доступом обратитесь к Главе 13.

Режим доступа транзакции определяет, будет ли транзакция только читать данные или будет и читать, и писать. По умолчанию подразумевается чтение/запись. В транзакции без записи

запрещаются следующие команды SQL: INSERT, UPDATE, DELETE и COPY FROM, если только целевая таблица не временная; любые команды CREATE, ALTER и DROP, а также COMMENT, GRANT, REVOKE, TRUNCATE; кроме того, запрещаются EXPLAIN ANALYZE и EXECUTE, если команда, которую они должны выполнить, относится к вышеперечисленным. Это высокоуровневое определение режима только для чтения, которое в принципе не исключает запись на диск.

Свойство DEFERRABLE оказывает влияние, только если транзакция находится также в режимах SERIALIZABLE и READ ONLY. Когда для транзакции установлены все три этих свойства, транзакция может быть заблокирована при первой попытке получить свой снимок данных, после чего она сможет выполняться без дополнительных усилий, обычных для режима SERIALIZABLE, и без риска привести к сбою сериализации или пострадать от него. Этот режим подходит для длительных операций, например для построения отчётов или резервного копирования.

Команда SET TRANSACTION SNAPSHOT позволяет выполнить новую транзакцию со *снимком данных*, который имеет уже существующая. Эта ранее созданная транзакция должна экспортировать этот снимок с помощью функции pg_export_snapshot (см. Подраздел 9.26.5). Эта функция возвращает идентификатор снимка, который и нужно передать команде SET TRANSACTION SNAPSHOT в качестве идентификатора импортируемого снимка. В данной команде этот идентификатор должен записываться в виде строковой константы, например '00000003-0000001B-1'. SET TRANSACTION SNAPSHOT можно выполнить только в начале транзакции, до первого запроса на выборку или изменение данных (SELECT, INSERT, DELETE, UPDATE, FETCH или COPY) в текущей транзакции. Более того, для транзакции уже должен быть установлен уровень изоляции SERIALIZABLE или REPEATABLE READ (в противном случае снимок будет сразу же потерян, так как на уровне READ COMMITTED для каждой команды делается новый снимок). Если импортирующая транзакция работает на уровне изоляции SERIALIZABLE, то транзакция, экспортирующая снимок, также должна работать на этом уровне. Кроме того, транзакции в режиме чтение/запись не могут импортировать снимок из транзакции в режиме «только чтение».

Замечания

Если команде SET TRANSACTION не предшествует START TRANSACTION или BEGIN, она выдаёт предупреждение и больше ничего не делает.

Без SET TRANSACTION можно обойтись, задав требуемые *режимы_транзакции* в операторах BEGIN или START TRANSACTION. Но для SET TRANSACTION SNAPSHOT такой возможности не предусмотрено.

Режимы транзакции для сеанса по умолчанию можно также задать или прочитать в конфигурационных переменных default_transaction_isolation, default_transaction_read_only и default_transaction_deferrable. (На практике, SET SESSION CHARACTERISTICS — это просто более многословная альтернатива изменению этих переменных командой SET.) Это значит, что значения этих переменных по умолчанию можно задать в файле конфигурации, с помощью команды ALTER DATABASE и т. д. За дополнительными сведениями обратитесь к Главе 18.

Режимы текущей транзакции можно также задать или прочитать в конфигурационных переменных default_transaction_isolation, transaction_read_only и transaction_deferrable. Присвоение значения одному из этих параметров равнозначна использованию SET TRANSACTION с теми же ограничениями по применению. Однако эти параметры нельзя задать в конфигурационном файле или каким-то ещё образом, кроме как непосредственно в SQL.

Примеры

Чтобы начать новую транзакцию со снимком данных, который получила уже существующая транзакция, его нужно сначала экспортировать из первой транзакции. При этом будет получен идентификатор снимка, например:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT pg_export_snapshot();
 pg_export_snapshot
---------------------
```

```
 00000003-0000001B-1
(1 row)
```

Затем этот идентификатор нужно передать команде `SET TRANSACTION SNAPSHOT` в начале новой транзакции:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION SNAPSHOT '00000003-0000001B-1';
```

Совместимость

Эти команды определены в стандарте SQL, за исключением режима транзакции `DEFERRABLE` и формы `SET TRANSACTION SNAPSHOT`, которые являются расширениями Postgres Pro.

В стандарте уровнем изоляции по умолчанию является `SERIALIZABLE`. В Postgres Pro уровнем по умолчанию обычно считается `READ COMMITTED`, но его можно изменить, как описано выше.

В стандарте SQL есть ещё одна характеристика транзакции, которую нельзя задать этими командами: размер диагностической области. Эта специфическая концепция встраиваемого SQL, поэтому в сервере Postgres Pro она не реализована.

Стандарт SQL требует, чтобы последовательные *режимы_транзакций* разделялись запятыми, но по историческим причинам Postgres Pro позволяет опустить запятые.

SHOW

SHOW — показать значение параметра времени выполнения

Синтаксис

```
SHOW имя
SHOW ALL
```

Описание

`SHOW` выводит текущие значения параметров времени выполнения. Эти переменные можно установить, воспользовавшись оператором `SET`, отредактировав файл конфигурации `postgresql.conf`, передав в переменной окружения `PGOPTIONS` (при использовании psql или приложения на базе libpq) либо в параметрах командной строки при запуске сервера `postgres`. За подробностями обратитесь к Главе 18.

Параметры

имя

Имя параметра времени выполнения. Доступные параметры описаны в Главе 18 и на странице справки SET. Кроме того, есть несколько параметров, которые можно просмотреть, но нельзя изменить:

`SERVER_VERSION`

Показывает номер версии сервера.

`SERVER_ENCODING`

Показывает кодировку набора символов на стороне сервера. В настоящее время этот параметр можно узнать, но нельзя изменить, так как кодировка определяется в момент создания базы данных.

`LC_COLLATE`

Показывает параметр локали базы данных, определяющий правило сортировки (порядок текстовых строк). В настоящее время этот параметр можно узнать, но нельзя изменить, так как он определяется в момент создания базы данных.

`LC_CTYPE`

Показывает параметр локали базы данных, определяющий классификацию символов. В настоящее время этот параметр можно узнать, но нельзя изменить, так как он определяется в момент создания базы данных.

`IS_SUPERUSER`

Возвращает true, если текущая роль обладает правами суперпользователя.

`ALL`

Показать значения всех конфигурационных параметров с описаниями.

Замечания

Ту же информацию выдаёт функция `current_setting`; см. Раздел 9.26. Кроме того, эту информацию можно получить через системное представление `pg_settings`.

Примеры

Просмотр текущего значения параметра `DateStyle`:

```
SHOW DateStyle;
 DateStyle
-----------
 ISO, MDY
(1 row)
```

Просмотр текущего значения параметра `geqo`:

```
SHOW geqo;
 geqo
------
 on
(1 row)
```

Просмотр всех параметров:

```
SHOW ALL;
            name          | setting |                 description

--------------------------+---------+-------------------------------------------------
 allow_system_table_mods | off     | Allows modifications of the structure of ...
    .
    .
    .
 xmloption               | content | Sets whether XML data in implicit parsing ...
 zero_damaged_pages      | off     | Continues processing past damaged page headers.
(196 rows)
```

Совместимость

Команда `SHOW` является расширением Postgres Pro.

См. также

SET, RESET

START TRANSACTION

START TRANSACTION — начать блок транзакции

Синтаксис

```
START TRANSACTION [ режим_транзакции [, ...] ]

Где режим_транзакции может быть следующим:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
 UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

Описание

Эта команда начинает новый блок транзакции. Если указан уровень изоляции, режим чтения/записи или допустимость откладывания транзакции, новая транзакция получит эти характеристики, как при выполнении команды SET TRANSACTION. Данная команда равнозначна команде BEGIN.

Параметры

За описанием параметров этого оператора обратитесь к SET TRANSACTION.

Совместимость

Согласно стандарту, выполнять `START TRANSACTION`, чтобы начать блок транзакции, необязательно: блок неявно начинает любая команда SQL. Поведение Postgres Pro можно представить как неявное выполнение `COMMIT` после каждой команды, которой не предшествует `START TRANSACTION` (или `BEGIN`), и поэтому такое поведение часто называется «автофиксацией». Другие реляционные СУБД тоже могут предлагать автофиксацию как удобную возможность.

Значение `DEFERRABLE` параметра *`режим_транзакции`* является языковым расширением Postgres Pro.

Стандарт SQL требует, чтобы последовательные *`режимы_транзакций`* разделялись запятыми, но по историческим причинам Postgres Pro позволяет опустить запятые.

См. также сведения о совместимости в описании SET TRANSACTION.

См. также

BEGIN, COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION

TRUNCATE

TRUNCATE — опустошить таблицу или набор таблиц

Синтаксис

```
TRUNCATE [ TABLE ] [ ONLY ] имя [ * ] [, ... ]
    [ RESTART IDENTITY | CONTINUE IDENTITY ] [ CASCADE | RESTRICT ]
```

Описание

Команда TRUNCATE быстро удаляет все строки из набора таблиц. Она действует так же, как безусловная команда DELETE для каждой таблицы, но гораздо быстрее, так как она фактически не сканирует таблицы. Более того, она немедленно высвобождает дисковое пространство, так что выполнять операцию VACUUM после неё не требуется. Наиболее полезна она для больших таблиц.

Параметры

имя

Имя таблицы (возможно, дополненное схемой), подлежащей опустошению. Если перед именем таблицы указано ONLY, очищается только заданная таблица. Без ONLY очищается и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно также добавить необязательное указание *, чтобы явно обозначить, что блокировка затрагивает и все дочерние таблицы.

RESTART IDENTITY

Автоматически перезапускать последовательности, связанные со столбцами опустошаемой таблицы.

CONTINUE IDENTITY

Не изменять значения последовательностей. Это поведение по умолчанию.

CASCADE

Автоматически опустошать все таблицы, ссылающиеся по внешнему ключу на заданные таблицы, или на таблицы, затронутые в результате действия CASCADE.

RESTRICT

Отказать в опустошении любых таблиц, на которые по внешнему ключу ссылаются другие таблицы, не перечисленные в этой команде. Это поведение по умолчанию.

Замечания

Чтобы опустошить таблицу, необходимо иметь право TRUNCATE для этой таблицы.

Команда TRUNCATE запрашивает блокировку ACCESS EXCLUSIVE для каждой таблицы, которую она обрабатывает. Когда указано RESTART IDENTITY, все последовательности, которые должны быть перезапущены, также блокируются исключительно. В случаях, когда требуется обеспечить параллельный доступ к таблице, следует использовать DELETE.

TRUNCATE нельзя использовать с таблицей, на которую по внешнему ключу ссылаются другие таблицы, если только и эти таблицы не опустошаются этой же командой. Проверка допустимости очистки в таких случаях потребовала бы сканирования таблицы, а главная идея данной команды в том, чтобы не делать этого. Для автоматической обработки всех зависимых таблиц можно использовать указание CASCADE — но будьте очень осторожны с ним, иначе вы можете потерять данные, которые не собирались удалять!

При выполнении TRUNCATE не срабатывают никакие триггеры ON DELETE, которые могут быть настроены для таблиц. Однако при этом срабатывают триггеры ON TRUNCATE. Если триггеры ON TRUNCATE определены для любых из этих таблиц, то все триггеры BEFORE TRUNCATE срабатывают до того, как происходит опустошение, а все триггеры AFTER TRUNCATE срабатывают после того, как завершается опустошение последней таблицы и все последовательности сбрасываются. Триггеры срабатывают по порядку обработки таблиц (сначала для таблиц, перечисленных в команде, затем для тех, что затрагиваются каскадно).

Команда TRUNCATE небезопасна с точки зрения MVCC. После опустошения таблицы она будет выглядеть пустой для параллельных транзакций, если они работают со снимком, полученным до опустошения. За подробностями обратитесь к Разделу 13.5.

TRUNCATE является надёжной транзакционной операцией в отношении данных в таблицах: опустошение будет безопасно отменено, если окружающая транзакция не будет зафиксирована.

С указанием RESTART IDENTITY подразумеваемые операции ALTER SEQUENCE RESTART также выполняются транзакционно; то есть, они будут отменены, если окружающая транзакция не будет зафиксирована. Учтите, что если до того, как транзакция отменится, будут выполнены какие-либо дополнительные операции с последовательностями, эффект этих операций также будет отменён, но не их влияние на значение currval(); то есть после транзакции currval() продолжит возвращать последнее значение последовательности, полученное внутри прерванной транзакции, хотя сама последовательность уже может быть несогласованной с ним. Подобным образом обычно ведёт себя currval() после сбоя транзакции.

TRUNCATE в настоящее время не поддерживается для сторонних таблиц. Из этого следует, что если у целевой таблицы есть дочерние таблицы, являющиеся сторонними, команда не будет выполнена.

Примеры

Опустошение таблиц bigtable и fattable:

```
TRUNCATE bigtable, fattable;
```

Та же операция и сброс всех связанных генераторов последовательностей:

```
TRUNCATE bigtable, fattable RESTART IDENTITY;
```

Опустошение таблицы othertable и каскадная обработка всех таблиц, ссылающихся на othertable по ограничениям внешнего ключа:

```
TRUNCATE othertable CASCADE;
```

Совместимость

Стандарт SQL:2008 включает команду TRUNCATE с синтаксисом TRUNCATE TABLE *имя_таблицы*. Предложения CONTINUE IDENTITY/RESTART IDENTITY также описаны в стандарте, но с небольшими отличиями, хотя их назначение похоже. Поведение этой команды при параллельных операциях, согласно стандарту, отчасти определяются реализацией, так что приведённые выше замечания при необходимости следует учитывать и сопоставлять с другими реализациями.

См. также

DELETE

UNLISTEN

UNLISTEN — прекратить ожидание уведомления

Синтаксис

```
UNLISTEN { канал | * }
```

Описание

`UNLISTEN` применяется для отмены существующей подписки на получение событий `NOTIFY`. `UNLISTEN` отменяет существующую подписку в текущем сеансе Postgres Pro на канал уведомлений с именем *канал*. Специальный знак `*` отменяет все подписки в текущем сеансе.

В описании NOTIFY использование `LISTEN` и `NOTIFY` рассматривается более подробно.

Параметры

канал

Имя канала уведомлений (любой идентификатор).

*

Отменяются все текущие подписки на уведомления для активного сеанса.

Замечания

Вы можете также попытаться отменить подписку на канал, на который не подписаны; предупреждений или ошибки при этом не будет.

`UNLISTEN *` автоматически выполняется в конце каждого сеанса.

Транзакция, выполнившая `UNLISTEN`, не может быть подготовлена для двухфазной фиксации.

Примеры

Подписка на получение события:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

Сразу после выполнения `UNLISTEN` последующие сообщения `NOTIFY` игнорируются:

```
UNLISTEN virtual;
NOTIFY virtual;
-- событие NOTIFY не поступает
```

Совместимость

Команда `UNLISTEN` отсутствует в стандарте SQL.

См. также

LISTEN, NOTIFY

UPDATE

UPDATE — изменить строки таблицы

Синтаксис

```
[ WITH [ RECURSIVE ] запрос_WITH [, ...] ]
UPDATE [ ONLY ] имя_таблицы [ * ] [ [ AS ] псевдоним ]
    SET { имя_столбца = { выражение | DEFAULT } |
          ( имя_столбца [, ...] ) = [ ROW ] ( { выражение | DEFAULT } [, ...] ) |
          ( имя_столбца [, ...] ) = ( вложенный_SELECT )
        } [, ...]
    [ FROM элемент_FROM [, ...] ]
    [ WHERE условие | WHERE CURRENT OF имя_курсора ]
    [ RETURNING * | выражение_результата [ [ AS ] имя_результата ] [, ...] ]
```

Описание

`UPDATE` изменяет значения указанных столбцов во всех строках, удовлетворяющих условию. В предложении `SET` должны указываться только те столбцы, которые будут изменены; столбцы, не изменяемые явно, сохраняют свои предыдущие значения.

Изменить строки в таблице, используя информацию из других таблиц в базе данных, можно двумя способами: применяя вложенные запросы или указав дополнительные таблицы в предложении `FROM`. Выбор предпочитаемого варианта зависит от конкретных обстоятельств.

Предложение `RETURNING` указывает, что команда `UPDATE` должна вычислить и возвратить значения для каждой фактически изменённой строки. Вычислить в нём можно любое выражение со столбцами целевой таблицы и/или столбцами других таблиц, упомянутых во `FROM`. При этом в выражении будут использоваться новые (изменённые) значения столбцов таблицы. Список `RETURNING` имеет тот же синтаксис, что и список результатов `SELECT`.

Для выполнения этой команды необходимо иметь право `UPDATE` для таблицы, или как минимум для столбцов, перечисленных в списке изменяемых. Также необходимо иметь право `SELECT` для всех столбцов, значения которых считываются в *выражениях* или *условии*.

Параметры

запрос_WITH

Предложение `WITH` позволяет задать один или несколько подзапросов, на которые затем можно ссылаться по имени в запросе `UPDATE`. Подробнее об этом см. Раздел 7.8 и SELECT.

имя_таблицы

Имя таблицы (возможно, дополненное схемой), строки которой будут изменены. Если перед именем таблицы добавлено `ONLY`, соответствующие строки изменяются только в указанной таблице. Без `ONLY` строки будут также изменены во всех таблицах, унаследованных от указанной. При желании, после имени таблицы можно указать `*`, чтобы явно обозначить, что операция затрагивает все дочерние таблицы.

псевдоним

Альтернативное имя целевой таблицы. Когда указывается это имя, оно полностью скрывает фактическое имя таблицы. Например, в запросе `UPDATE foo AS f` дополнительные компоненты оператора `UPDATE` должны обращаться к целевой таблице по имени `f`, а не `foo`.

имя_столбца

Имя столбца в таблице *имя_таблицы*. Имя столбца при необходимости может быть дополнено именем вложенного поля или индексом массива. Имя таблицы добавлять к имени целевого столбца не нужно — например, запись `UPDATE table_name SET table_name.col = 1` ошибочна.

выражение

Выражение, результат которого присваивается столбцу. В этом выражении можно использовать предыдущие значения этого и других столбцов таблицы.

DEFAULT

Присвоить столбцу значение по умолчанию (это может быть NULL, если для столбца не определено некоторое выражение по умолчанию).

вложенный_SELECT

Подзапрос SELECT, выдающий столько выходных столбцов, сколько перечислено в предшествующем ему списке столбцов в скобках. При выполнении этого подзапроса должна быть получена максимум одна строка. Если он выдаёт одну строку, значения столбцов в нём присваиваются целевым столбцам; если же он не возвращает строку, целевым столбцам присваивается NULL. Этот подзапрос может обращаться к предыдущим значениям текущей изменяемой строки в таблице.

элемент_FROM

Табличное выражение, позволяющее обращаться в условии WHERE и выражениях новых данных к столбцам других таблиц. В нём используется тот же синтаксис, что и в предложении «Предложение FROM» оператора SELECT; например, вы можете определить псевдоним для таблицы. Имя целевой таблицы повторять в предложении FROM нужно, только если вы хотите определить замкнутое соединение (в этом случае для данного имени должен определяться псевдоним).

условие

Выражение, возвращающее значение типа boolean. Изменены будут только те стоки, для которых это выражение возвращает true.

имя_курсора

Имя курсора, который будет использоваться в условии WHERE CURRENT OF. С таким условием будет изменена строка, выбранная из этого курсора последней. Курсор должен образовываться запросом, не применяющим группировку, к целевой таблице команды UPDATE. Заметьте, что WHERE CURRENT OF нельзя задать вместе с логическим условием. За дополнительными сведениями об использовании курсоров с WHERE CURRENT OF обратитесь к DECLARE.

выражение_результата

Выражение, которое будет вычисляться и возвращаться командой UPDATE после изменения каждой строки. В этом выражении можно использовать имена любых столбцов таблицы *имя_таблицы* или таблиц, перечисленных в списке FROM. Чтобы получить все столбцы, достаточно написать *.

имя_результата

Имя, назначаемое возвращаемому столбцу.

Выводимая информация

В случае успешного завершения, UPDATE возвращает метку команды в виде

UPDATE *число*

Здесь *число* обозначает количество изменённых строк, включая те подлежащие изменению строки, значения в которых не были изменены. Заметьте, что это число может быть меньше количества строк, удовлетворяющих *условию*, когда изменения отменяются триггером BEFORE UPDATE. Если *число* равно 0, данный запрос не изменил ни одной строки (это не считается ошибкой).

Если команда UPDATE содержит предложение RETURNING, её результат будет похож на результат оператора SELECT (с теми же столбцами и значениями, что содержатся в списке RETURNING), полученный для строк, изменённых этой командой.

Замечания

Когда присутствует предложение FROM, целевая таблица по сути соединяется с таблицами, перечисленными в *элементе_FROM*, и каждая выходная строка соединения представляет операцию изменения для целевой таблицы. Применяя предложение FROM, необходимо обеспечить, чтобы соединение выдавало максимум одну выходную строку для каждой строки, которую нужно изменить. Другими словами, целевая строка не должна соединяться с более чем одной строкой из других таблиц. Если это условие нарушается, только одна из строк соединения будет использоваться для изменения целевой строки, но какая именно, предсказать нельзя.

Из-за этой неопределённости надёжнее ссылаться на другие таблицы только в подзапросах, хотя такие запросы часто хуже читаются и работают медленнее, чем соединение.

В секционированной таблице строка при изменении может перестать удовлетворять ограничению содержащей её секции. Ввиду отсутствия средств для переноса строки в другую секцию, соответствующую новому значению ключа секционирования, в этой ситуации произойдёт ошибка. Это также возможно при непосредственном изменении данных в секции.

Примеры

Изменение слова Drama на Dramatic в столбце kind таблицы films:

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

Изменение значений температуры и сброс уровня осадков к значению по умолчанию в одной строке таблицы weather:

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

Выполнение той же операции с получением изменённых записей:

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03'
  RETURNING temp_lo, temp_hi, prcp;
```

Такое же изменение с применением альтернативного синтаксиса со списком столбцов:

```
UPDATE weather SET (temp_lo, temp_hi, prcp) = (temp_lo+1, temp_lo+15, DEFAULT)
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

Увеличение счётчика продаж для менеджера, занимающегося компанией Acme Corporation, с применением предложения FROM:

```
UPDATE employees SET sales_count = sales_count + 1 FROM accounts
  WHERE accounts.name = 'Acme Corporation'
  AND employees.id = accounts.sales_person;
```

Выполнение той же операции, с вложенным запросом в предложении WHERE:

```
UPDATE employees SET sales_count = sales_count + 1 WHERE id =
  (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation');
```

Изменение имени контакта в таблице счетов (это должно быть имя назначенного менеджера по продажам):

```
UPDATE accounts SET (contact_first_name, contact_last_name) =
    (SELECT first_name, last_name FROM salesmen
     WHERE salesmen.id = accounts.sales_id);
```

Подобный результат можно получить, применив соединение:

```
UPDATE accounts SET contact_first_name = first_name,
                    contact_last_name = last_name
  FROM salesmen WHERE salesmen.id = accounts.sales_id;
```

Однако если `salesmen.id` — не уникальный ключ, второй запрос может давать непредсказуемые результаты, тогда как первый запрос гарантированно выдаст ошибку, если найдётся несколько записей с одним `id`. Кроме того, если соответствующая запись `accounts.sales_id` не найдётся, первый запрос запишет в поля имени NULL, а второй вовсе не изменит строку.

Обновление статистики в сводной таблице в соответствии с текущими данными:

```
UPDATE summary s SET (sum_x, sum_y, avg_x, avg_y) =
    (SELECT sum(x), sum(y), avg(x), avg(y) FROM data d
     WHERE d.group_id = s.group_id);
```

Попытка добавить новый продукт вместе с количеством. Если такая запись уже существует, вместо этого увеличить количество данного продукта в существующей записи. Чтобы реализовать этот подход, не откатывая всю транзакцию, можно использовать точки сохранения:

```
BEGIN;
-- другие операции
SAVEPOINT sp1;
INSERT INTO wines VALUES('Chateau Lafite 2003', '24');
-- Предполагая, что здесь возникает ошибка из-за нарушения уникальности ключа,
-- мы выполняем следующие команды:
ROLLBACK TO sp1;
UPDATE wines SET stock = stock + 24 WHERE winename = 'Chateau Lafite 2003';
-- Продолжение других операций и в завершение...
COMMIT;
```

Изменение столбца `kind` таблицы `films` в строке, на которой в данный момент находится курсор `c_films`:

```
UPDATE films SET kind = 'Dramatic' WHERE CURRENT OF c_films;
```

Совместимость

Эта команда соответствует стандарту SQL, за исключением предложений `FROM` и `RETURNING`, которые являются расширениями Postgres Pro, как и возможность применять `WITH` с `UPDATE`.

В некоторых других СУБД также поддерживается дополнительное предложение `FROM`, но предполагается, что целевая таблица должна ещё раз упоминаться в этом предложении. Postgres Pro воспринимает предложение `FROM` не так, поэтому будьте внимательны, портируя приложения, которые используют это расширение языка.

Согласно стандарту, исходным значением для вложенного списка имён столбцов в скобках может быть любое выражение, выдающее строку с нужным количеством столбцов. Postgres Pro принимает в качестве этого значения только конструктор строки или вложенный `SELECT`. Изменяемое значение отдельного столбца можно обозначать словом `DEFAULT` в конструкторе строки, но не внутри вложенного `SELECT`.

VACUUM

VACUUM — провести сборку мусора и, возможно, проанализировать базу данных

Синтаксис

```
VACUUM [ ( { FULL | FREEZE | VERBOSE | ANALYZE | DISABLE_PAGE_SKIPPING } [, ...] ) ]
  [ имя_таблицы [ (имя_столбца [, ...] ) ] ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ имя_таблицы ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ имя_таблицы [ (имя_столбца
  [, ...] ) ] ]
```

Описание

VACUUM высвобождает пространство, занимаемое «мёртвыми» кортежами. При обычных операциях Postgres Pro кортежи, удалённые или устаревшие в результате обновления, физически не удаляются из таблицы; они сохраняются в ней, пока не будет выполнена команда VACUUM. Таким образом, периодически необходимо выполнять VACUUM, особенно для часто изменяемых таблиц.

Без параметра команда VACUUM обрабатывает все таблицы в текущей базе данных, которые может очистить текущий пользователь. Если в параметре передаётся имя таблицы, VACUUM обрабатывает только эту таблицу.

VACUUM ANALYZE выполняет очистку (VACUUM), а затем анализ (ANALYZE) всех указанных таблиц. Это удобная комбинация для регулярного обслуживания БД. За дополнительной информацией об анализе обратитесь к описанию ANALYZE.

Простая команда VACUUM (без FULL) только высвобождает пространство и делает его доступным для повторного использования. Эта форма команды может работать параллельно с обычными операциями чтения и записи таблицы, так она не требует исключительной блокировки. Однако освобождённое место не возвращается операционной системе (в большинстве случаев); оно просто остаётся доступным для размещения данных этой же таблицы. VACUUM FULL переписывает всё содержимое таблицы в новый файл на диске, не содержащий ничего лишнего, что позволяет возвратить неиспользованное пространство операционной системе. Эта форма работает намного медленнее и запрашивает блокировку ACCESS EXCLUSIVE для каждой обрабатываемой таблицы.

Когда список параметров заключается в скобки, параметры могут быть записаны в любом порядке. Без скобок параметры должны указываться именно в том порядке, который показан выше. Синтаксис со скобками появился в PostgreSQL 9.0; вариант записи без скобок считается устаревшим.

Параметры

FULL

Выбирает режим «полной» очистки, который может освободить больше пространства, но выполняется гораздо дольше и запрашивает исключительную блокировку таблицы. Этот режим также требует дополнительное место на диске, так как он записывает новую копию таблицы и не освобождает старую до завершения операции. Обычно это следует использовать, только когда требуется высвободить значительный объём пространства, выделенного таблице.

FREEZE

Выбирает агрессивную «заморозку» кортежей. Добавление указания FREEZE равносильно выполнению команды VACUUM с параметрами vacuum_freeze_min_age и vacuum_freeze_table_age, равными нулю. Агрессивная заморозка всегда выполняется при перезаписи таблицы, поэтому в режиме FULL это указание избыточно.

VERBOSE

Выводит подробный отчёт об очистке для каждой таблицы.

ANALYZE

Обновляет статистику, которую использует планировщик для выбора наиболее эффективного способа выполнения запроса.

DISABLE_PAGE_SKIPPING

Обычно VACUUM пропускает страницы, учитывая карту видимости. Страницы, на которых, судя по карте, все кортежи заморожены, можно пропускать всегда, а страницы, в которых все кортежи видны всем транзакциям, могут обрабатываться только при агрессивной очистке. Более того, за исключением агрессивной очистки, некоторые страницы можно пропускать, чтобы не ждать, пока другие сеансы закончат их использовать. Этот параметр отключает пропуск страниц и рассчитан для использования только когда целостность карты видимости вызывает подозрения, что возможно при аппаратных или программных сбоях, приводящих к разрушению БД.

имя_таблицы

Имя (возможно, дополненное схемой) определённой таблицы, подлежащей очистке. Если оно отсутствует, очистке подвергаются все обычные таблицы и материализованные представления в текущей базе данных. Если указанная таблица является секционированной, очищаться будут все её конечные секции.

имя_столбца

Имя столбца, подлежащего анализу. По умолчанию анализируются все столбцы. Если указан список столбцов, подразумевается операция ANALYZE.

Выводимая информация

С указанием VERBOSE команда VACUUM выдаёт сообщения о процессе очистки, отмечая текущую обрабатываемую таблицу. Также она выводит различные статистические сведения о таблицах.

Замечания

Чтобы очистить таблицу, обычно нужно быть владельцем этой таблицы или суперпользователем. Однако владельцам баз данных также разрешено сжимать все таблицы в своих базах, за исключением общих каталогов. (Ограничение в отношении общих каталогов означает, что настоящую глобальную команду VACUUM может выполнить только суперпользователь.) VACUUM при обработке пропускает все таблицы, на очистку которых текущий пользователь не имеет прав.

VACUUM нельзя выполнять внутри блока транзакции.

Для таблиц с индексами GIN, VACUUM (в любой форме) также завершает все ожидающие операции добавления в индекс, перемещая записи индекса из очереди в соответствующие места в основной структуре индекса GIN. За подробностями обратитесь к Подразделу 60.4.1.

Мы рекомендуем очищать активные производственные базы данных достаточно часто (как минимум, каждую ночь), чтобы избавляться от «мёртвых» строк. После добавления или удаления большого числа строк может быть хорошей идеей выполнить команду VACUUM ANALYZE для каждой затронутой таблицы. При этом результаты всех последних изменений будут отражены в системных каталогах, что позволит планировщику запросов Postgres Pro принимать более эффективные решения при планировании.

Режим FULL не рекомендуется для обычного применения, но в некоторых случаях он бывает полезен. Например, когда были удалены или изменены почти все строки таблицы, может возникнуть желание физически сжать её, чтобы освободить место на диске и ускорить сканирование этой таблицы. Чаще всего VACUUM FULL сжимает таблицу более эффективно, чем обычный VACUUM.

VACUUM создаёт значительную нагрузку на подсистему ввода/вывода, что может отрицательно сказаться на производительности других активных сеансов. Поэтому иногда полезно использовать возможность задержки очистки с учётом её стоимости. За подробностями обратитесь к Подразделу 18.4.4.

Postgres Pro включает средство «автоочистки», которое позволяет автоматизировать регулярную очистку. Чтобы узнать больше об автоматической и ручной очистке, обратитесь к Разделу 23.1.

Примеры

Очистка одной таблицы onek, проведение её анализа для оптимизатора и печать подробного отчёта о действиях операции очистки:

```
VACUUM (VERBOSE, ANALYZE) onek;
```

Совместимость

Оператор VACUUM отсутствует в стандарте SQL.

См. также

vacuumdb, Подраздел 18.4.4, Подраздел 23.1.6

VALUES

VALUES — вычислить набор строк

Синтаксис

```
VALUES ( выражение [, ...] ) [, ...]
    [ ORDER BY выражение_сортировки [ ASC | DESC | USING оператор ] [, ...] ]
    [ LIMIT { число | ALL } ]
    [ OFFSET начало [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ число ] { ROW | ROWS } ONLY ]
```

Описание

`VALUES` вычисляет значение строки или множество значений строк, заданное выражениями. Чаще всего эта команда используется для формирования «таблицы констант» в большой команде, но её можно использовать и отдельно.

Когда указывается больше, чем одна строка, все строки должны иметь одинаковое количество элементов. Типы данных результирующих столбцов таблицы определяются в результате совмещения явных и неявных типов выражений, заданных для этих столбцов, по тем же правилам, что и в `UNION` (см. Раздел 10.5).

В составе других команд синтаксис допускает использование `VALUES` везде, где допускается `SELECT`. Так как грамматически она воспринимается как `SELECT`, с командой `VALUES` можно использовать предложения `ORDER BY`, `LIMIT` (или равнозначное `FETCH FIRST`) и `OFFSET`.

Параметры

выражение

Константа или выражение, которое вычисляется и вставляется в указанное место результирующей таблицы (множества строк). В списке `VALUES`, находящемся на верхнем уровне оператора `INSERT`, *выражение* может быть заменено словом `DEFAULT`, указывающим, что в целевой столбец должно быть вставлено значение этого столбца по умолчанию. Когда `VALUES` употребляется в других контекстах, указание `DEFAULT` использовать нельзя.

выражение_сортировки

Выражение или целочисленная константа, указывающая, как должны сортироваться строки результата. Это выражение может обращаться к столбцам результата `VALUES` по именам `column1`, `column2` и т. д. За дополнительными подробностями обратитесь к Подразделу «Предложение `ORDER BY`».

оператор

Оператор сортировки. За подробностями обратитесь к Подразделу «Предложение `ORDER BY`».

число

Максимальное число строк, которое должно быть возвращено. За подробностями обратитесь к Подразделу «Предложение `LIMIT`».

начало

Число строк, которые должны быть пропущены, прежде чем начнётся выдача строк. За подробностями обратитесь к Подразделу «Предложение `LIMIT`».

Замечания

Следует избегать составления списков `VALUES` с очень большим количеством строк, так как при этом можно столкнуться с нехваткой памяти или снижением производительности. Применение

VALUES в команде INSERT — особый случай (так как ожидаемые типы столбцов становятся известны из целевой таблицы команды INSERT и их не надо вычислять, сканируя весь список VALUES), так что в этом контексте можно работать с гораздо более объёмными списками, чем в других.

Примеры

Простейшая команда VALUES:

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

Эта команда выдаст таблицу из двух столбцов и трёх строк. По сути она равнозначна запросу:

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

Более типично использование VALUES в составе большей команды SQL. Чаще всего она применяется в INSERT:

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

В контексте INSERT список VALUES может содержать слово DEFAULT, указывающее, что в данном месте вместо некоторого значения должно использоваться значение столбца по умолчанию:

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes'),
    ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama', DEFAULT);
```

VALUES также может применяться там, где можно написать вложенный SELECT, например в предложении FROM:

```
SELECT f.*
  FROM films f, (VALUES('MGM', 'Horror'), ('UA', 'Sci-Fi')) AS t (studio, kind)
  WHERE f.studio = t.studio AND f.kind = t.kind;

UPDATE employees SET salary = salary * v.increase
  FROM (VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v (depno, target, increase)
  WHERE employees.depno = v.depno AND employees.sales >= v.target;
```

Заметьте, что когда VALUES используется в предложении FROM, предложение AS становится обязательным, так же, как и для SELECT. При этом не требуется указывать в AS имена всех столбцов, но это рекомендуется делать. (По умолчанию Postgres Pro даёт столбцам VALUES имена column1, column2 и т. д., но в других СУБД имена могут быть другими.)

Когда VALUES используется в команде INSERT, значения автоматически приводятся к типу данных соответствующего целевого столбца. Когда оно используется в других контекстах, может потребоваться указать нужный тип данных. Если все записи представлены строковыми константами в кавычках, достаточно привести к нужному типу значения в первой строке, чтобы задать тип для всех:

```
SELECT * FROM machines
WHERE ip_address IN (VALUES('192.168.0.1'::inet), ('192.168.0.10'), ('192.168.1.43'));
```

Подсказка

Для простых проверок на включение IN лучше полагаться на форму IN со списком скаляров, чем записывать запрос VALUES, как показано выше. Список скаляров проще записать и обрабатывается он зачастую более эффективно.

Совместимость

`VALUES` соответствует стандарту SQL. Указания `LIMIT` и `OFFSET` являются расширениями Postgres Pro; см. также SELECT.

См. также

INSERT, SELECT

WAITLSN

WAITLSN — ожидать воспроизведения целевого LSN

Синтаксис

```
WAITLSN 'LSN' [ INFINITELY ]
WAITLSN 'LSN' TIMEOUT время_ожидания
WAITLSN 'LSN' NOWAIT
```

Описание

`WAITLSN` предоставляет простой межпроцессный механизм ожидания целевого LSN (последовательного номера записи в журнале) на ведомых узлах в схемах синхронной репликации ведущий-ведомый Postgres Pro. Команда `WAITLSN` ждёт воспроизведения заданного LSN. По умолчанию время ожидания неограниченно. Ожидание можно прервать, нажав `Ctrl+C` или остановив сервер `postgres`. Вы также можете задать время ожидания, добавив указание `TIMEOUT`, или проверить состояние целевого LSN без ожидания, воспользовавшись указанием `NOWAIT`.

> **Важно**
>
> Оператор `WAITLSN` считается устаревшим.

Параметры

LSN

Указывает последовательный номер записи в журнале, воспроизведение которой ожидается.

INFINITELY

Ожидать воспроизведения целевого LSN на ведомом сервере неограниченное время. Это необязательный параметр, подтверждающий поведение по умолчанию.

TIMEOUT *время_ожидания*

Ограничивает время ожидания воспроизведения LSN. В параметре *время_ожидания* указывается целое число миллисекунд.

NOWAIT

С этим указанием команда сообщает, был ли уже воспроизведён заданный LSN, без ожидания.

Примеры

Выполнение `WAITLSN` в psql с ограничением времени до 10000 миллисекунд:

```
WAITLSN '0/3F07A6B1' TIMEOUT 10000;
NOTICE:  LSN is not reached. Try to increase wait time.
LSN reached
-------------
 f
(1 row)
```

Ожидание воспроизведения указанного LSN:

```
WAITLSN '0/3F07A611';
LSN reached
-------------
```

```
 t
(1 row)
```

Ограничение времени ожидания LSN до 500000 миллисекунд и отмена команды:

```
WAITLSN '0/3F0FF791' TIMEOUT 500000;
^CCancel request sent
NOTICE:  LSN is not reached. Try to increase wait time.
ERROR:  canceling statement due to user request
 LSN reached
-------------
 f
(1 row)
```

Совместимость

Оператор `WAITLSN` отсутствует в стандарте SQL.

Клиентские приложения Postgres Pro

Раздел описывает клиентские приложения и утилиты Postgres Pro. Некоторые из описанных приложений требуют особые привилегии. Основной отличительной особенностью этих приложений является возможность исполнения на любом компьютере, независимо от расположения сервера баз данных.

Имя пользователя и базы данных передаются из командной строки на сервер без изменения регистра — все пробельные и специальные символы необходимо экранировать с помощью кавычек. Имена таблиц и другие идентификаторы передаются регистр-независимо, за исключением отдельно описанных ситуаций, где может требоваться экранирование.

clusterdb

clusterdb — кластеризовать базу данных Postgres Pro

Синтаксис

`clusterdb` [*параметр-подключения*...] [`--verbose` | `-v`] [`--table` | `-t` *таблица*] ... [*имя_бд*]

`clusterdb` [*параметр-подключения*...] [`--verbose` | `-v`] `--all` | `-a`

Описание

clusterdb это приложение для повторной кластеризации таблиц базы данных Postgres Pro. Утилита находит ранее кластеризованные таблицы и проводит операцию на основании последнего использованного индекса. Затрагиваются лишь ранее кластеризованные таблицы.

clusterdb — это обёртка для SQL-команды CLUSTER. Кластеризация баз данных с её помощью по сути не отличается от выполнения того же действия при обращении к серверу другими способами.

Параметры

clusterdb принимает следующие аргументы командной строки:

`-a`
`--all`

Кластеризовать все базы данных.

`[-d]` *имя_бд*
`[--dbname=]`*имя_бд*

Указывает имя базы данных для кластеризации, когда не используется параметр `-a`/`--all`. Если это указание отсутствует, имя базы определяется переменной окружения `PGDATABASE`. Если эта переменная не установлена, именем базы будет имя пользователя, указанное для подключения. В аргументе *имя_бд* может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

`-e`
`--echo`

Вывести команды к серверу, генерируемые при выполнении clusterdb.

`-q`
`--quiet`

Подавлять вывод сообщений о прогрессе выполнения.

`-t` *таблица*
`--table=`*таблица*

Кластеризовать *таблицу*. Возможно множественное использование параметра `-t`.

`-v`
`--verbose`

Вывести подробную информацию во время процесса.

`-V`
`--version`

Вывести версию clusterdb и прервать дальнейшее выполнение.

`-?`
`--help`

Вывести справку по аргументам команды clusterdb.

clusterdb также принимает из командной строки параметры подключения:

`-h` *сервер*
`--host=`*сервер*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *порт*
`--port=`*порт*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

`-U` *имя_пользователя*
`--username=`*имя_пользователя*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как clusterdb запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, clusterdb лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

`--maintenance-db=`*имя_бд*

Указывает имя базы данных, к которой будет выполняться подключение для определения подлежащих кластеризации баз данных, когда используется ключ `-a`/`--all`. Если это имя не указано, будет выбрана база `postgres`, а если она не существует — `template1`. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке. Кроме того, все параметры в строке подключения, за исключением имени базы, будут использоваться и при подключении к другим базам данных.

Переменные окружения

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 31.14).

Диагностика

В случае возникновения трудностей, обратитесь к CLUSTER и psql. Переменные окружения и параметры подключения по умолчанию libpq будут применены при запуске утилиты, это следует учитывать при диагностике.

Примеры

Для кластеризации базы данных `test`:

```
$ clusterdb test
```

Для кластеризации отдельной таблицы `foo` базы данных `xyzzy`:

```
$ clusterdb --table=foo xyzzy
```

См. также

CLUSTER

createdb

createdb — создать базу данных Postgres Pro

Синтаксис

`createdb` [*параметр-подключения*...] [*параметр*...] [*имя_бд* [*описание*]]

Описание

createdb создаёт базу данных Postgres Pro.

Чаще всего пользователь, выполняющий эту команду, назначается владельцем создаваемой базы данных. Однако можно указать владельца явным образом с помощью флага `-O`, если у текущего пользователя достаточно привилегий.

createdb это обёртка для SQL-команды CREATE DATABASE. Создание баз данных с её помощью по сути не отличается от выполнения того же действия при обращении к серверу другими способами.

Параметры

createdb принимает в качестве аргументов:

имя_бд

Указывает имя создаваемой базы. Имя должно быть уникальным в рамках кластера Postgres Pro. По умолчанию в качестве имени базы данных берётся имя текущего системного пользователя.

описание

Добавляет комментарий к создаваемой базе.

`-D` *табличное_пространство*
`--tablespace=`*табличное_пространство*

Указывает табличное пространство, используемое по умолчанию. Имя пространства обрабатывается аналогично идентификаторам, заключённым в двойные кавычки.

`-e`
`--echo`

Вывести команды к серверу, генерируемые при выполнении createdb.

`-E` *кодировка*
`--encoding=`*кодировка*

Указывает кодировку базы данных. Поддерживаемые сервером Postgres Pro кодировки описаны в Подразделе 22.3.1.

`-l` *локаль*`[@`*провайдер*`]`
`--locale=`*локаль*`[@`*провайдер*`]`

Указывает локаль базы данных. Имеет эффект одновременно установленных флагов `--lc-collate` и `--lc-ctype`.

Кроме того, вы можете задать провайдер основного правила сортировки после символа `@`. Допустимые варианты этого указания: `icu` и `libc`. За подробностями обратитесь к Подразделу 22.2.2.

`--lc-collate=`*локаль*`[@`*провайдер*`]`

Устанавливает параметр LC_COLLATE для базы данных.

Кроме того, вы можете задать провайдер основного правила сортировки после символа `@`. Допустимые варианты этого указания: `icu` и `libc`. За подробностями обратитесь к Подразделу 22.2.2.

`--lc-ctype=`*локаль*

Устанавливает параметр LC_CTYPE для базы данных.

`-O` *владелец*
`--owner=`*владелец*

Указывает пользователя в качестве владельца создаваемой базы. Имя пользователя обрабатывается аналогично идентификаторам, заключённым в двойные кавычки.

`-T` *шаблон*
`--template=`*шаблон*

Указывает шаблон, на основе которого будет создана база данных. Имя шаблона обрабатывается аналогично идентификаторам, заключённым в двойные кавычки.

`-V`
`--version`

Вывести версию createdb и прервать дальнейшее исполнение.

`-?`
`--help`

Вывести помощь по команде createdb и прервать выполнение.

Флаги `-D`, `-l`, `-E`, `-O` и `-T` по назначению соответствуют флагам SQL-команды CREATE DATABASE.

createdb также принимает из командной строки параметры подключения:

`-h` *сервер*
`--host=`*сервер*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *порт*
`--port=`*порт*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

`-U` *имя_пользователя*
`--username=`*имя_пользователя*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как createdb запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, createdb лишний раз

подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

`--maintenance-db=`*`имя_бд`*

Указывает имя опорной базы данных, к которой будет произведено подключение для создания новой. Если имя не указано, будет выбрана база `postgres`, а если она не существует — `template1`. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

Переменные окружения

`PGDATABASE`

Если установлено и не переопределено в командной строке, задаёт имя создаваемой базы данных.

`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию. `PGUSER` указывает имя пользователя при создании базы данных, если не указано явно в командной строке или в переменной окружения `PGDATABASE`.

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 31.14).

Диагностика

В случае возникновения трудностей обратитесь к CREATE DATABASE и psql. При диагностике нужно учитывать, что при запуске утилиты используются значения переменных окружения и параметров подключения по умолчанию libpq.

Примеры

Создать базу данных `demo` на сервере, используемом по умолчанию, можно так:

```
$ createdb demo
```

Создать базу `demo` на сервере `eden`, порт 5000, из шаблонной базы `template0` можно такой командой командной строки, за которой стоит следующая команда SQL:

```
$ createdb -p 5000 -h eden -T template0 -e demo
CREATE DATABASE demo TEMPLATE template0;
```

См. также

dropdb, CREATE DATABASE

createuser

createuser — создать новую учётную запись Postgres Pro

Синтаксис

`createuser` [*параметр-подключения*...] [*параметр*...] [*имя_пользователя*]

Описание

createuser создаёт нового пользователя Postgres Pro, а если точнее — роль. Лишь суперпользователь и пользователи с привилегией `CREATEROLE` могут создавать новые роли, таким образом, createuser должна запускаться от их лица.

Чтобы создать дополнительного суперпользователя, необходимо подключиться от имени существующего, одного лишь права `CREATEROLE` недостаточно. Поскольку суперпользователи могут обходить все ограничения доступа в базе данных, к назначению этих полномочий не следует относиться легкомысленно.

createuser — это обёртка для SQL-команды CREATE ROLE. Создание пользователей с её помощью по сути не отличается от выполнения того же действия при обращении к серверу другими способами.

Параметры

createuser принимает следующие аргументы:

имя_пользователя

Задаёт имя создаваемого пользователя Postgres Pro. Это имя должно отличаться от имён всех существующих ролей в данной инсталляции Postgres Pro.

`-c` *номер*
`--connection-limit=`*номер*

Устанавливает максимальное допустимое количество соединений для создаваемого пользователя. По умолчанию ограничение в количестве соединений отсутствует.

`-d`
`--createdb`

Разрешает новому пользователю создавать базы данных.

`-D`
`--no-createdb`

Запрещает новому пользователю создавать базы данных. Это поведение по умолчанию.

`-e`
`--echo`

Вывести команды к серверу, генерируемые при выполнении createuser.

`-E`
`--encrypted`

Параметр является устаревшим, но в целях совместимости ещё работает.

`-g` *role*
`--role=`*role*

Указывает роль, к которой будет добавлена текущая роль в качестве члена группы. Допускается множественное использование флага `-g`.

`-i`
`--inherit`

Создаваемая роль автоматически унаследует права ролей, в которые она включается. Это поведение по умолчанию.

`-I`
`--no-inherit`

Роль не будет наследовать права ролей, в которые она включается.

`--interactive`

Запросить имя для создаваемого пользователя, а также значения для флагов `-d`/`-D`, `-r`/`-R`, `-s`/`-S`, если они явно не указаны в командной строке. До версии PostgreSQL 9.1 включительно это было поведением по умолчанию.

`-l`
`--login`

Новый пользователь сможет подключаться к серверу (то есть его имя может быть идентификатором начального пользователя сеанса). Это свойство по умолчанию.

`-L`
`--no-login`

Новый пользователь не сможет подключаться к серверу. (Роль без права входа на сервер тем не менее полезна для управления разрешениями в базе данных.)

`-P`
`--pwprompt`

Если флаг указан, то createuser запросит пароль для создаваемого пользователя. Если не планируется аутентификация по паролю, то пароль можно не устанавливать.

`-r`
`--createrole`

Разрешает новому пользователю создавать другие роли, что означает наделение привилегией `CREATEROLE`.

`-R`
`--no-createrole`

Запрещает пользователю создавать новые роли. Это поведение по умолчанию.

`-s`
`--superuser`

Создаваемая роль будет иметь права суперпользователя.

`-S`
`--no-superuser`

Новый пользователь не будет суперпользователем. Это поведение по умолчанию.

`-V`
`--version`

Вывести версию createuser и завершить выполнение.

`--replication`

Создаваемый пользователь будет наделён правом `REPLICATION`. Это рассмотрено подробнее в документации по CREATE ROLE.

`--no-replication`

Создаваемый пользователь не будет иметь привилегии `REPLICATION`. Это рассмотрено подробнее в документации по CREATE ROLE.

`-?`
`--help`

Вывести помощь по команде createuser.

createuser также принимает из командной строки параметры подключения:

`-h` *сервер*
`--host=`*сервер*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *порт*
`--port=`*порт*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

`-U` *имя_пользователя*
`--username=`*имя_пользователя*

Имя пользователя для подключения (не имя создаваемого пользователя).

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как createuser запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, createuser лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

Переменные окружения

`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 31.14).

Диагностика

В случае возникновения трудностей, обратитесь к CREATE ROLE и psql. Переменные окружения и параметры подключения по умолчанию libpq будут применены при запуске утилиты, это следует учитывать при диагностике.

Примеры

Чтобы создать роль `joe` на сервере, используемом по умолчанию:

```
$ createuser joe
```

Чтобы создать роль `joe` на сервере, используемом по умолчанию, с запросом дополнительных параметров:

```
$ createuser --interactive joe
Назначить роль суперпользователем? (y/n) n
Разрешить новой роли создавать базы данных? (y/n) n
Разрешить новой роли создавать другие роли? (y/n) n
```

Чтобы создать того же пользователя `joe` с явно заданными атрибутами, подключившись к компьютеру `eden`, порту 5000:

```
$ createuser -h eden -p 5000 -S -D -R -e joe
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

Чтобы создать роль `joe` с правами суперпользователя и предустановленным паролем:

```
$ createuser -P -s -e joe
Введите пароль для новой роли: xyzzy
Повторите его: xyzzy
CREATE ROLE joe PASSWORD 'md5b5f5ba1a423792b526f799ae4eb3d59e' SUPERUSER CREATEDB
 CREATEROLE INHERIT LOGIN;
```

В приведённом примере введённый пароль отображается лишь для отражения сути, на деле же он не выводится на экран. Как можно видеть, он шифруется прежде чем передаётся в команде клиенту.

См. также

dropuser, CREATE ROLE

dropdb

dropdb — удалить базу данных Postgres Pro

Синтаксис

`dropdb` [*параметр-подключения*...] [*параметр*...] *имя_бд*

Описание

dropdb удаляет ранее созданную базу данных Postgres Pro, и должна выполняться от имени суперпользователя или её владельца.

dropdb это обёртка для SQL-команды DROP DATABASE. Удаление баз данных с её помощью по сути не отличается от выполнения того же действия при обращении к серверу другими способами.

Параметры

dropdb принимает в качестве аргументов:

имя_бд

Указывает имя удаляемой базы данных.

`-e`
`--echo`

Вывести команды к серверу, генерируемые при выполнении dropdb.

`-i`
`--interactive`

Выводит вопрос о подтверждении перед удалением.

`-V`
`--version`

Выводит версию dropdb.

`--if-exists`

Не считать ошибкой, если база данных не существует. В этом случае будет выдано замечание.

`-?`
`--help`

Вывести справку по команде dropdb.

dropdb также принимает из командной строки параметры подключения:

`-h` *сервер*
`--host=`*сервер*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *порт*
`--port=`*порт*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

`-U` *имя_пользователя*
`--username=`*имя_пользователя*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как dropdb запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, dropdb лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

`--maintenance-db=`*имя_бд*

Указывает имя опорной базы данных, к которой будет произведено подключение для удаления целевой. Если имя не указано, будет выбрана база `postgres`, а если она не существует (или именно она и удаляется) — `template1`. Здесь может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

Переменные окружения

`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 31.14).

Диагностика

В случае возникновения трудностей, обратитесь к DROP DATABASE и psql. При диагностике следует учесть, что при запуске утилиты также применяются переменные окружения и параметры подключения по умолчанию libpq.

Примеры

Для удаления базы данных `demo` на сервере, используемом по умолчанию:

```
$ dropdb demo
```

Для удаления базы данных `demo` на сервере `eden`, слушающим подключения на порту 5000, в интерактивном режиме и выводом запросов к серверу:

```
$ dropdb -p 5000 -h eden -i -e demo
База данных "demo" будет удалена навсегда.
Продолжить? (y/n) y
DROP DATABASE demo;
```

См. также

createdb, DROP DATABASE

dropuser

dropuser — удалить учётную запись пользователя Postgres Pro

Синтаксис

`dropuser` [*параметр-подключения*...] [*параметр*...] [*имя_пользователя*]

Описание

dropuser удаляет ранее созданного пользователя Postgres Pro. Лишь суперпользователь или пользователь с привилегией `CREATEROLE` могут удалять пользователей Postgres Pro. Необходимо быть суперпользователем, чтобы удалить учётную запись другого суперпользователя.

dropuser это обёртка для SQL-команды DROP ROLE. Удаление пользователей с её помощью по сути не отличается от выполнения того же действия при обращении к серверу другими способами.

Параметры

dropuser принимает в качестве аргументов:

имя_пользователя

Указывает имя удаляемой роли Postgres Pro. Если передан флаг `-i`/`--interactive`, а имя не указано в параметрах команды, его необходимо будет ввести интерактивно.

`-e`
`--echo`

Вывести команды к серверу, генерируемые при выполнении dropuser.

`-i`
`--interactive`

Вывести подтверждение об удалении роли, и запросить её имя, если оно не указано в параметрах команды.

`-V`
`--version`

Вывести версию dropuser.

`--if-exists`

Перехватить ошибку, если пользователь не существует. В этом случае вместо ошибки будет выведено информационное сообщение.

`-?`
`--help`

Вывести справку по команде dropuser.

dropuser также принимает из командной строки параметры подключения:

`-h` *сервер*
`--host=`*сервер*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *порт*
`--port=`*порт*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

`-U` *имя_пользователя*
`--username=`*имя_пользователя*

Имя пользователя, под которым производится текущее подключение к базе.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как dropuser запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, dropuser лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

Переменные окружения

`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 31.14).

Диагностика

В случае возникновения трудностей, обратитесь к DROP ROLE и psql. При диагностике следует учесть, что при запуске утилиты также применяются переменные окружения и параметры подключения по умолчанию libpq.

Примеры

Чтобы удалить роль `joe` на сервере, используемом по умолчанию:

```
$ dropuser joe
```

Чтобы удалить роль `joe` на сервере `eden`, слушающем подключения на порту 5000, в интерактивном режиме и с выводом выполняемых команд:

```
$ dropuser -p 5000 -h eden -i -e joe
Роль "joe" будет удалена навсегда.
Продолжить? (y/n) y
DROP ROLE joe;
```

См. также

createuser, DROP ROLE

ecpg

ecpg — встроенный C-препроцессор SQL

Синтаксис

ecpg [*параметр*...] *файл*...

Описание

ecpg — это встроенный SQL препроцессор для программ, написанных на языке C. Он преобразует программы на C, содержащие SQL-выражения, заменяя их вызовами встроенных функций. Получаемые на выходе файлы можно затем скомпилировать и скомпоновать.

ecpg преобразует каждый файл, переданный в параметрах, в соответствующий файл на C. Если имя входного файла указано без расширения, подразумевается .pgc. Указанное имя, но с расширением .c, становится по умолчанию именем выходного файла. Переопределить имя выходного файла можно, воспользовавшись параметром -o.

Если в качестве имени входного файла указано -, ecpg читает программу с устройства стандартного ввода (и выводит результат в стандартный вывод, если не задан параметр -o).

Данный раздел не содержит описания встроенного SQL-языка. Для более подробной информации см. Главу 33.

Параметры

ecpg принимает в качестве аргументов:

-c

Автоматически генерировать код, написанный на языке C, из кода SQL. Сейчас это справедливо для EXEC SQL TYPE.

-C *режим*

Установить режим совместимости; *режим* может принимать значение INFORMIX или INFORMIX_SE.

-D *символ*

Определить символ начала команд C-препроцессора.

-h

Обрабатывать заголовочные файлы. Когда добавляется этот параметр, расширением выходного файла становится не .c, а .h, и расширением входного файла по умолчанию считается не .pgc, а .pgh. Кроме этого с данным параметром подразумевается -c.

-i

Также разбирать и системные включения.

-I *каталог*

Указать дополнительный путь включаемых файлов, используемый при выполнении EXEC SQL INCLUDE. По умолчанию используются . (текущий каталог), /usr/local/include, каталог, задаваемый при компиляции Postgres Pro (обычно — /usr/local/pgsql/include), и /usr/include, в порядке, как это перечислено.

-o *имя_файла*

Задаёт *имя файла*, в который программа ecpg должна выводить результат. Указание -o - направляет результат в устройство стандартного вывода.

`-r` *параметр*

Определяет поведение времени выполнения. *Флаг* может принимать следующие значения:

`no_indicator`

Использовать специальные символы для представления значений null. Исторически некоторые базы данных используют такой подход.

`prepare`

Сформировать подготовленные выражения. libecpg сформирует кеш подготовленных выражений и будет использовать их при необходимости повторно. В случае переполнения кеша, libecpg освободит память за счёт вытеснения наименее используемых выражений.

`questionmarks`

Разрешает использовать знак вопроса в качестве аргумента подстановки в целях совместимости. Ранее это было поведением по умолчанию.

`-t`

Включить автоматическую фиксацию транзакций. В этом режиме каждая SQL-команда будет автоматически фиксироваться, пока не будет явно включена в блок транзакции. В режиме по умолчанию команды фиксируются лишь при явном вызове `EXEC SQL COMMIT`.

`-v`

Вывести информацию о версии, а также путях поиска включаемых файлов.

`--version`

Вывести версию ecpg.

`-?`
`--help`

Вывести справку по команде ecpg.

Замечания

При компиляции полученных файлов, компилятор должен иметь возможность найти заголовочные файлы ECPG в каталоге включений Postgres Pro. Для этого можно использовать флаг `-I` во время компиляции, например, `-I/usr/local/pgsql/include`.

Программы на C со встроенным SQL необходимо скомпоновать с библиотекой `libecpg`, например, используя флаг компоновщика `-L/usr/local/pgsql/lib -lecpg`.

Имена каталогов, подходящих для установки, можно найти в разделе pg_config.

Примеры

Если имеется исходный файл на C `prog1.pgc` со встроенным SQL, можно создать исполняемую программу, используя следующую последовательность команд:

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecpg
```

pg_basebackup

pg_basebackup — создать резервную копию кластера Postgres Pro

Синтаксис

`pg_basebackup` [*параметр*...]

Описание

pg_basebackup предназначен для создания резервных копий работающего кластера баз данных Postgres Pro. Процедура создания копии не влияет на работу других клиентов. Полученные копии могут использоваться для обеих стратегий восстановления — на заданный момент в прошлом (см. Раздел 24.3) и в качестве отправной точки для ведомого сервера при реализации трансляции файлов или потоковой репликации (см. Раздел 25.2).

pg_basebackup создаёт бинарную копию файлов кластера, контролируя режим создания копии автоматически. Резервные копии всегда создаются для кластера целиком и невозможно создать копию для какой-либо сущности базы отдельно. Для этой цели можно использовать, например, утилиту pg_dump.

Копия создаётся через обычное подключение к Postgres Pro, и при этом используется протокол репликации. Подключение должно осуществляться от лица суперпользователя или пользователя с правом `REPLICATION` (см. Раздел 20.2), а в `pg_hba.conf` должно быть прописано подключение для репликации. Значение max_wal_senders на сервере должно быть достаточно большим, чтобы допускать минимум ещё одно подключение для копирования и одно для трансляции WAL (если она используется).

Можно запустить одновременно несколько команд `pg_basebackup`, но с точки зрения производительности лучше делать всего одну копию одновременно, а затем копировать получаемый результат.

С помощью pg_basebackup можно получить базовую копию не только на ведущем, но и на ведомом сервере. Для этого на ведомом сервере необходимо разрешить соединения репликации (параметры `max_wal_senders` и hot_standby, а также настроить аутентификацию компьютера). При этом на ведущем необходимо включить full_page_writes.

Заметьте, что при копировании с ведомого сервера есть некоторые ограничения:

- Файл истории резервного копирования в целевом кластере баз данных не создаётся.
- pg_basebackup не может принудительно переключить ведомый сервер на новый файл WAL в конце копирования. Поэтому, если используется режим `-X none` и активность записи на ведущем сервере низкая, pg_basebackup может довольно долго ждать переключения и архивирования последнего файла WAL, необходимого для полноты копии. В этом случае может иметь смысл выполнить на ведущем сервере `pg_switch_wal` для немедленного переключения файла WAL.
- Если ведомый сервер переключается в роль ведущего в процессе копирования, копирование прерывается.
- Все необходимые для резервной копии WAL-записи должны содержать полные страницы, для чего нужно включить режим `full_page_writes` на ведущем и не использовать в `archive_command` такие утилиты, как pg_compresslog, которые могут удалить записанные полные страницы из WAL.

Параметры

Описанные далее аргументы командной строки влияют на размещение и формат вывода.

`-D` *каталог*
`--pgdata=`*каталог*

Целевой каталог для записи данных. pg_basebackup создаст его и родительские, если необходимо. Каталог может быть создан заранее, но должен быть пустым, иначе возникнет ошибка.

Если резервирование работает в режиме `tar`, а имя каталога имеет значение `-` (тире), то tar-файл будет писаться в `stdout`.

Этот флаг является обязательным.

`-F` *формат*
`--format=`*формат*

Устанавливает формат вывода. *формат* может принимать следующие значения:

`p`
`plain`

Записывает выводимые данные в обычные файлы, сохраняя структуру текущих каталогов данных и табличных пространств. Если в кластере нет дополнительных табличных пространств, вся база будет помещена в заданный каталог. Иначе основной каталог хранения данных будет помещён в целевой каталог, а все остальные табличные пространства — в те же абсолютные пути, в которых они располагаются на исходном сервере. (Чтобы изменить эти пути, воспользуйтесь параметром `--tablespace-mapping`).

Это формат по умолчанию.

`t`
`tar`

Записывает в целевой каталог файлы в формате tar. Основной каталог хранения данных будет писаться в файл `base.tar`, а табличные пространства — в файлы, именованные в соответствии с их OID.

Если имя целевого каталога задано как `-` (тире), то данные будут писаться в стандартный вывод, что позволяет, например, использовать gzip. Это возможно лишь когда не применяются дополнительные табличные пространства и не используется трансляция WAL.

`-r` *скорость_передачи*
`--max-rate=`*скорость_передачи*

Максимальная скорость передачи данных с сервера. Значение задаётся в Кб/с. Для установки значения в мегабайтах, можно использовать суффикс `M`. Также допустим суффикс `k`, но он не принципиален. Допустимые значения лежат в рамках между 32 Кб/с и 1024 Мб/с.

Служит для снижения влияния на производительность сервера со стороны работающего pg_basebackup.

Этот параметр всегда оказывает влияние на передачу каталога данных, а на передачу файлов WAL он влияет, только если выбран метод передачи `fetch`.

`-R`
`--write-recovery-conf`

Записать минимальный файл `recovery.conf` в каталог вывода (или базовый архивный файл при использовании формата tar) для упрощения настройки ведомого сервера. В файл `recovery.conf` будут записаны параметры соединения и, если указан, слот репликации, который использует pg_basebackup, так что впоследствии при потоковой репликации будут использоваться те же параметры.

`-S` *имя_слота*
`--slot=`*имя_слота*

Этот параметр может применяться только вместе с `-X stream`. Он устанавливает использование заданного слота репликации при потоковой передаче WAL. Если базовая копия предназначена для использования на ведомом сервере с потоковой репликацией, затем должен использоваться слот с тем же именем в `recovery.conf`. Тем самым гарантируется, что сервер не удалит никакие необходимые данные WAL после того, как базовая копия будет получена, и до того, как начнётся потоковая репликация.

Если этот ключ не указан и сервер поддерживает временные слоты репликации (они появились в версии 10), для трансляции WAL автоматически используется временный слот репликации.

`--no-slot`

Этот ключ предотвращает создание временного слота репликации во время резервного копирования, даже если это поддерживается сервером.

Временные слоты репликации создаются по умолчанию, если при трансляции журнала имя слота не задаётся в параметре `-S`.

Основное предназначение этого ключа в том, чтобы можно было сделать базовую резервную копию, когда на сервере не хватает свободных слотов репликации. Использование слотов репликации почти всегда предпочтительнее, так как при этом предотвращается удаление во время резервного копирования необходимых файлов WAL с сервера.

`-T` *старый_каталог*`=`*новый_каталог*
`--tablespace-mapping=`*старый_каталог*`=`*новый_каталог*

Переместить табличное пространство из *старого_каталога* в *новый_каталог* в процессе копирования. Чтобы перемещение произошло, в параметре *старый_каталог* должен задаваться в точности путь табличного пространства, как он определён. (Но не будет ошибкой, если табличного пространства, на которое указывает *старый_каталог*, в архиве не окажется.) И *старый_каталог*, и *новый_каталог* должны задаваться абсолютными путями. Если в пути встречается символ `=`, его необходимо экранировать обратной косой чертой. Этот параметр можно добавить несколько раз для нескольких табличных пространств. См. примеры ниже.

Если табличное пространство перемещается таким способом, символические ссылки внутри основного каталога хранения данных также приводятся в соответствие с новым местоположением. Таким образом, для экземпляра сервера подготавливается новый каталог данных, в котором все табличные пространства оказываются в новом расположении.

В настоящее время этот параметр работает только с обычным форматом вывода; если выбран формат tar, параметр игнорируется.

`--waldir=`*каталог_wal*

Указывает размещение каталога хранения журнала предзаписи. Задаваемый в параметре *каталог_wal* путь должен быть абсолютным. Каталог с журналом предзаписи можно задать только при создании копии в простом режиме.

`-X` *метод*
`--wal-method=`*метод*

Включает все необходимые файлы журналов предзаписи (файлы WAL) в резервную копию. В том числе включаются все журналы предзаписи, сгенерированные в процессе создания резервной копии. Если только не выбран метод `none`, главный процесс БД может быть запущен непосредственно с восстановленным каталогом, без обращения к дополнительному архиву журналов; таким образом будет получена полностью самодостаточная резервная копия.

Для сбора журналов предзаписи поддерживаются следующие методы:

`n`
`none`

Не включать журнал предзаписи в резервную копию.

`f`
`fetch`

Файлы журнала предзаписи собираются в конце процесса копирования. Таким образом необходимо установить достаточно большое значение параметра wal_keep_segments, чтобы избежать преждевременного удаления файлов журнала. В случае удаления файлов до завершения процесса копирования возникнет ошибка, а копия будет непригодной к использованию.

Когда используется формат tar, файлы журнала предзаписи записываются в файл `base.tar`.

`s`
`stream`

Передавать журнал предзаписи в процессе создания резервной копии. При этом открывается второе соединение к серверу, по которому будет передаваться журнал предзаписи, одновременно с созданием резервной копии. Таким образом будут использоваться два подключения из разрешённых параметром max_wal_senders. И если клиент будет успевать получать журнал предзаписи, ведущему серверу не потребуется хранить дополнительные файлы журнала.

Когда используется формат tar, файлы журнала предзаписи сохраняются в отдельном файле с именем `pg_wal.tar` (если версия сервера ниже 10, файл будет называться `pg_xlog.tar`).

Это значение по умолчанию.

`-z`
`--gzip`

Включает gzip-сжатие выводимого tar-файла с уровнем компрессии по умолчанию. Сжатие поддерживается только для формата tar, при этом ко всем именам файлов tar добавляется суффикс `.gz`.

`-Z` *уровень*
`--compress=`*уровень*

Включает gzip-сжатие выводимого tar-файла и задаёт уровень сжатия от 0 (без сжатия) до 9 (максимальное сжатие). Сжатие поддерживается только для формата tar, при этом ко всем именам файлов tar добавляется суффикс `.gz`.

Описанные далее аргументы командной строки влияют на генерацию резервной копии и ход выполнения приложения.

`-c` *fast*|*spread*
`--checkpoint=`*fast*|*spread*

Устанавливает режим контрольных точек: fast (быстрый) или spread (протяжённый, по умолчанию). Подробнее см. Подраздел 24.3.3.

`-l` *метка*
`--label=`*метка*

Устанавливает метку для созданной резервной копии. Если не указана, то по умолчанию будет использовано значение «`pg_basebackup base backup`».

`-n`
`--no-clean`

По умолчанию, когда программа `pg_basebackup` прерывается с ошибкой, она удаляет все каталоги, которые она могла создать, прежде чем обнаружила, что не может завершить

задание (например, каталог данных и каталог журнала предзаписи). Данный ключ отключает эту очистку и тем самым полезен для отладки.

Заметьте, что каталоги табличных пространств не очищаются в любом случае.

`-P`
`--progress`

Включает отчёт о прогрессе. Если этот режим включён, то во время создания копии будет передаваться примерный процент выполнения. Так как данные в базе могут меняться во время копирования, это значение будет лишь приближённым и может достигать не точно `100%`. В частности, когда в копию включается журнал WAL, конечный размер невозможно предсказать заранее, и в этом случае ожидаемый конечный размер будет увеличиваться, превысив ориентировочный полный размер без WAL.

Если режим включён, то процесс копирования начнется с перечисления размеров всей базы, а затем продолжится отправкой непосредственно данных. Это может немного увеличить время операции, в частности, пройдёт больше времени до начала передачи данных.

`-N`
`--no-sync`

По умолчанию `pg_basebackup` ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром `pg_basebackup` завершается немедленно, то есть выполняется быстрее, но в случае неожиданного сбоя операционной системы резервная копия может оказаться испорченной. Вообще этот параметр предназначен прежде всего для тестирования, для производственной среды он не подходит.

`-v`
`--verbose`

Включает режим подробного вывода. Будет выводится некоторая дополнительная информация при начале и завершении, а также имена обрабатываемых файлов, если включён отчёт о прогрессе.

Далее описаны параметры управления подключением.

`-d` *`строка_подключения`*
`--dbname=`*`строка_подключения`*

Указывает параметры подключения к серверу в формате строки подключения; они будут переопределять любые одноимённые параметры, заданные в командной строке.

Параметр называется `--dbname` для согласованности с другими клиентскими приложениями, но так как pg_basebackup не подключается к какой-либо конкретной базе, это имя в строке подключения игнорируется.

`-h` *`сервер`*
`--host=`*`сервер`*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета. Значение по умолчанию берётся из переменной окружения `PGHOST`, если она установлена. В противном случае выполняется подключение к Unix-сокету.

`-p` *`порт`*
`--port=`*`порт`*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной окружения `PGPORT`, если она установлена, либо числом, заданным при компиляции.

`-s` *interval*
`--status-interval=`*interval*

Указывает интервал в секундах между отправкой пакетов статуса, отправляемых на сервер. Это позволяет упростить мониторинг прогресса. Чтобы выключить периодическое обновление статуса, необходимо установить значение в ноль. При этом обновление будет отправляться по запросу сервера для избежания отсоединения по истечению времени. Значение по умолчанию составляет 10 секунд.

`-U` *имя_пользователя*
`--username=`*имя_пользователя*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как pg_basebackup запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, pg_basebackup лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

Другие флаги:

`-V`
`--version`

Вывести версию pg_basebackup.

`-?`
`--help`

Вывести справку по команде pg_basebackup.

Переменные окружения

Как и большинство других утилит Postgres Pro, приложение также использует переменные окружения, поддерживаемые libpq (см. Раздел 31.14).

Замечания

Прежде чем начнётся копирование, на сервере с копируемой базой необходимо выполнить контрольную точку. И если копирование запускается без ключа `--checkpoint=fast`, это может занять некоторое время, в течение которого pg_basebackup не будет проявлять никакой активности.

Резервная копия будет включать в себя все файлы каталога хранения данных и табличных пространств, а также конфигурационные файлы и прочие файлы, размещённые в каталоге данных, за исключением определённых временных файлов, принадлежащих Postgres Pro. Однако копируются лишь простые файлы и каталоги, кроме них, сохраняются только символические ссылки на табличные пространства. Символические ссылки, указывающие на определённые каталоги, известные Postgres Pro, копируются как пустые каталоги. Другие символические

ссылки и файлы спецустройств игнорируются. За дополнительными подробностями обратитесь к Разделу 50.4.

Если не указан параметр `--tablespace-mapping`, табличные пространства в простом формате будут копироваться в тот же путь, который они имеют на сервере. Поэтому при наличии табличных пространств создать базовую копию в простом формате на том же сервере не удастся, так как копия будет направлена в те же каталоги, где располагаются исходные табличные пространства.

Когда применяется режим формата tar, пользователь должен позаботиться о том, чтобы все архивы tar были распакованы до запуска сервера Postgres Pro. Если имеются дополнительные табличные пространства, архивы tar для них должны быть распакованы в правильные каталоги. В таком случае для этих табличных пространств сервером будут созданы символические ссылки, согласно содержимому файла `tablespace_map`, включённого в архив `base.tar`.

pg_basebackup совместим с серверами той же или более младших версий, но не ниже 9.1. Однако режим трансляции WAL (`-X stream`) поддерживается с версиями сервера не ниже 9.3, а режим формата tar (`--format=tar`) текущей версии совместим только с версиями сервера не ниже 9.5.

Примеры

Создание резервной копии сервера `mydbserver` и сохранение её в локальном каталоге `/usr/local/pgsql/data`:

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
```

Создание резервной копии локального сервера в отдельных сжатых файлах tar для каждого табличного пространства и сохранение их в каталоге `backup` с индикатором прогресса в процессе выполнения:

```
$ pg_basebackup -D backup -Ft -z -P
```

Создание резервной копии локальной базы данных с одним табличным пространством и сжатие её с помощью bzip2:

```
$ pg_basebackup -D - -Ft -X fetch | bzip2 > backup.tar.bz2
```

(Эта команда прервётся с ошибкой, если в базе данных будет несколько табличных пространств.)

Создание резервной копии локальной базы данных с перемещением табличного пространства `/opt/ts` в `./backup/ts`:

```
$ pg_basebackup -D backup/data -T /opt/ts=$(pwd)/backup/ts
```

См. также

pg_dump

pgbench

pgbench — запустить тест производительности Postgres Pro

Синтаксис

`pgbench -i` [*параметр*...] [*имя_бд*]

`pgbench` [*параметр*...] [*имя_бд*]

Описание

pgbench — это простая программа для запуска тестов производительности Postgres Pro. Она многократно выполняет одну последовательность команд, возможно в параллельных сеансах базы данных, а затем вычисляет среднюю скорость транзакций (число транзакций в секунду). По умолчанию pgbench тестирует сценарий, примерно соответствующий TPC-B, который состоит из пяти команд `SELECT`, `UPDATE` и `INSERT` в одной транзакции. Однако вы можете легко протестировать и другие сценарии, написав собственные скрипты транзакций.

Типичный вывод pgbench выглядит так:

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 10
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
tps = 85.184871 (including connections establishing)
tps = 85.296346 (excluding connections establishing)
```

В первых шести строках выводятся значения некоторых самых важных параметров. В следующей строке показывается количество выполненных и запланированных транзакций (это будет произведение числа клиентов и числа транзакций для одного клиента); эти количества будут равны, если только выполнение не завершится досрочно. (В режиме `-T` выводится только число фактически выполненных транзакций.) В последних двух строках показывается число транзакций в секунду, подсчитанное с учётом и без учёта времени установления подключения к серверу.

Для запускаемого по умолчанию теста типа TPC-B требуется предварительно подготовить определённые таблицы. Чтобы создать и наполнить эти таблицы, следует запустить pgbench с ключом `-i` (инициализировать). (Если вы применяете нестандартный скрипт, это не требуется, но тем не менее нужно подготовить конфигурацию, нужную вашему тесту.) Запуск инициализации выглядит так:

`pgbench -i [ `*`другие-параметры`*` ] `*`имя_базы`*

где *имя_базы* — имя уже существующей базы, в которой будет проводиться тест. (Чтобы указать, как подключиться к серверу баз данных, вы также можете добавить параметры `-h`, `-p` и/или `-U`.)

> **Внимание**
>
> `pgbench -i` создаёт четыре таблицы `pgbench_accounts`, `pgbench_branches`, `pgbench_history` и `pgbench_tellers`, предварительно уничтожая существующие таблицы с этими именами. Если вы вдруг используете эти имена в своей базе данных, обязательно переключитесь на другую базу!

С «коэффициентом масштаба», по умолчанию равным 1, эти таблицы изначально содержат такое количество строк:

```
table                    # of rows
---------------------------------
pgbench_branches         1
pgbench_tellers          10
pgbench_accounts         100000
pgbench_history          0
```

Эти числа можно (и в большинстве случаев даже нужно) увеличить, воспользовавшись параметром `-s` (коэффициент масштаба). При этом также может быть полезен ключ `-F` (фактор заполнения).

Подготовив требуемую конфигурацию, можно запустить тест производительности командой без `-i`, то есть:

```
pgbench [ параметры ] имя_базы
```

Практически во всех случаях, чтобы получить полезные результаты, необходимо передать какие-либо дополнительные параметры. Наиболее важные параметры: `-c` (число клиентов), `-t` (число транзакций), `-T` (длительность) и `-f` (файл со скриптом). Полный список параметров приведён ниже.

Параметры

Следующий список разделён на три подраздела: одни параметры используются при инициализации базы данных, другие при проведении тестирования, а третьи в обоих случаях.

Параметры инициализации

pgbench принимает следующие аргументы командной строки для инициализации:

`-i`
`--initialize`

Требуется для вызова режима инициализации.

`-F` *фактор_заполнения*
`--fillfactor=`*фактор_заполнения*

Создать таблицы `pgbench_accounts`, `pgbench_tellers` и `pgbench_branches` с заданным фактором заполнения. Значение по умолчанию — 100.

`-n`
`--no-vacuum`

Не выполнять очистку после инициализации.

`-q`
`--quiet`

Переключить вывод в немногословный режим, когда выводится только одно сообщение о прогрессе в 5 секунд. В режиме по умолчанию одно сообщение выводится на каждые 100000 строк, при этом за секунду обычно выводится довольно много строк (особенно на хорошем оборудовании).

`-s` *коэффициент_масштаба*
`--scale=`*коэффициент_масштаба*

Умножить число генерируемых строк на заданный коэффициент. Например, с ключом `-s 100` в таблицу `pgbench_accounts` будут записаны 10 000 000 строк. Значение по умолчанию — 1. При коэффициенте, равном 20 000 или больше, столбцы, содержащие идентификаторы счетов (столбцы `aid`), перейдут к большим целым числам (типу `bigint`), чтобы в них могли уместиться все возможные значения идентификаторов.

`--foreign-keys`

Создать ограничения внешних ключей между стандартными таблицами.

`--index-tablespace=`*`табл_пространство_индексов`*

Создать индексы в указанном табличном пространстве, а не в пространстве по умолчанию.

`--tablespace=`*`табличное_пространство`*

Создать таблицы в указанном табличном пространстве, а не в пространстве по умолчанию.

`--unlogged-tables`

Создать все таблицы как нежурналируемые, а не как постоянные таблицы.

Параметры тестирования производительности

pgbench принимает следующие аргументы командной строки для тестирования производительности:

`-b` *`имя_скрипта[@вес]`*
`--builtin=`*`имя_скрипта[@вес]`*

Добавляет в список скриптов, которые будут выполняться, указанный встроенный скрипт. В число встроенных скриптов входят `tpcb-like`, `simple-update` и `select-only`. Также принимаются однозначные начала их имён. Со специальным именем `list` программа выводит список встроенных скриптов и немедленно завершается.

Дополнительно можно задать целочисленный вес после `@`, меняющий вероятность выбора этого скрипта относительно других. По умолчанию вес считается равным 1. Подробности следуют ниже.

`-c` *`клиенты`*
`--client=`*`клиенты`*

Число имитируемых клиентов, то есть число одновременных сеансов базы данных. Значение по умолчанию — 1.

`-C`
`--connect`

Устанавливать новое подключение для каждой транзакции вместо одного для каждого клиента. Это полезно для оценивания издержек подключений.

`-d`
`--debug`

Выводить отладочные сообщения.

`-D` *`имя_переменной=значение`*
`--define=`*`имя_переменной=значение`*

Определить переменную для пользовательского скрипта (см. ниже). Параметр `-D` может добавляться неоднократно.

`-f` *`имя_файла[@вес]`*
`--file=`*`имя_файла[@вес]`*

Добавить в список выполняемых скриптов скрипт транзакции из файла *`имя_файла`*.

Дополнительно можно задать целочисленный вес после `@`, меняющий вероятность выбора этого скрипта относительно других. По умолчанию вес считается равным 1. (Если вам нужно передать имя скрипта, содержащее символ `@`, добавьте к такому имени вес, чтобы исключить неоднозначность прочтения, например `filen@me@1`.) Подробности следуют ниже.

`-j` *`потоки`*
`--jobs=`*`потоки`*

Число рабочих потоков в pgbench. Использовать нескольких потоков может быть полезно на многопроцессорных компьютерах. Клиенты распределяются по доступным потокам равномерно, насколько это возможно. Значение по умолчанию — 1.

`-l`
`--log`

Записать информацию о каждой транзакции в файл протокола. Подробности описаны ниже.

`-L` *предел*
`--latency-limit=`*предел*

Транзакции, продолжающиеся дольше указанного *предела* (в миллисекундах), подсчитываются и отмечаются отдельно, как *опаздывающие*.

В режиме ограничения скорости (`--rate=...`) транзакции, которые отстают от графика более чем на заданный *предел* (в мс) и поэтому никак не могут уложиться в отведённый интервал, не передаются серверу вовсе. Они подсчитываются и отмечаются отдельно как *пропущенные*.

`-M` *режим_запросов*
`--protocol=`*режим_запросов*

Протокол, выбираемый для передачи запросов на сервер:

- `simple`: использовать протокол простых запросов.
- `extended`: использовать протокол расширенных запросов.
- `prepared`: использовать протокол расширенных запросов с подготовленными операторами.

По умолчанию выбирается протокол простых запросов. (За подробностями обратитесь к Главе 50.)

`-n`
`--no-vacuum`

Не производить очистку таблиц перед запуском теста. Этот параметр *необходим*, если вы применяете собственный сценарий, не затрагивающий стандартные таблицы `pgbench_accounts`, `pgbench_branches`, `pgbench_history` и `pgbench_tellers`.

`-N`
`--skip-some-updates`

Запустить встроенный упрощённый скрипт simple-update. Краткий вариант записи `-b simple-update`.

`-P` *сек*
`--progress=`*сек*

Выводить отчёт о прогрессе через заданное число секунд (*сек*). Выдаваемый отчёт включает время, прошедшее с момента запуска, скорость (в tps) с момента предыдущего отчёта, а также среднее время ожидания транзакций и стандартное отклонение. В режиме ограничения скорости (`-R`) время ожидания вычисляется относительно назначенного времени запуска транзакции, а не фактического времени её начала, так что оно включает и среднее время отставания от графика.

`-r`
`--report-latencies`

Выводить по завершении тестировании средняя время ожидания операторов (время выполнения с точки зрения клиента) для каждой команды. Подробности описаны ниже.

`-R` *скорость передачи*
`--rate=`*скорость передачи*

Выполнять транзакции, ориентируясь на заданную скорость, а не максимально быстро (по умолчанию). Скорость задаётся в транзакциях в секунду. Если заданная скорость превышает максимально возможную, это ограничение скорости не повлияет на результаты.

Для получения нужной скорости транзакции запускаются со случайными задержками, имеющими распределение Пуассона. При этом запланированное время запуска отсчитывается

от начального времени, а не от завершения предыдущей транзакции. Это означает, что если какие-то транзакции отстанут от изначально рассчитанного времени завершения, всё же возможно, что последующие нагонят график.

В режиме ограничения скорости время ожидания транзакций, выводимое по итогам тестирования, вычисляется, исходя из запланированного времени запуска, так что в него входит время, которое очередная транзакция должна была ждать завершения предыдущей транзакции. Это время называется временем отклонения от графика, и его среднее и максимальное значения выводятся отдельно. Время ожидания транзакций с момента их фактического запуска, то есть время, потраченное на выполнение транзакций в базе данных, можно получить, если вычесть время отклонения от графика из времени ожидания транзакций.

Если ограничение `--latency-limit` задаётся вместе с `--rate`, транзакция может заведомо не вписываться в отведённое ей время, если предыдущая транзакция завершится слишком поздно, так как ожидаемое время окончания транзакции отсчитывается от времени запуска по графику. Такие транзакции не передаются серверу, а пропускаются и подсчитываются отдельно.

Большое значение отклонения от графика свидетельствует о том, что система не успевает выполнять транзакции с заданной скоростью и выбранным числом клиентов и потоков. Когда среднее время ожидания транзакции превышает запланированный интервал между транзакциями, каждая последующая транзакция будет отставать от графика, и чем дольше будет выполняться тестирование, тем больше будет отставание. Когда это наблюдается, нужно уменьшить скорость транзакций.

`-s` *коэффициент_масштаба*
`--scale=`*коэффициент_масштаба*

Показать заданный коэффициент масштаба в выводе pgbench. Для встроенных тестов это не требуется; корректный коэффициент масштаба будет получен в результате подсчёта строк в таблице `pgbench_branches`. Однако при использовании только нестандартных тестов (запускаемых с ключом `-f`) без этого параметра в качестве коэффициента масштаба будет выводиться 1.

`-S`
`--select-only`

Запустить встроенный скрипт select-only (только выборка). Краткий вариант записи `-b select-only`.

`-t` *транзакции*
`--transactions=`*транзакции*

Число транзакций, которые будут выполняться каждым клиентом (по умолчанию 10).

`-T` *секунды*
`--time=`*секунды*

Выполнять тест с ограничением по времени (в секундах), а не по числу транзакций для каждого клиента. Параметры `-t` и `-T` являются взаимоисключающими.

`-v`
`--vacuum-all`

Очищать все четыре стандартные таблицы перед запуском теста. Без параметров `-n` и `-v` pgbench будет очищать от старых записей таблицы `pgbench_tellers` и `pgbench_branches`, а также опустошать `pgbench_history`.

`--aggregate-interval=`*секунды*

Длительность интервала агрегации (в секундах). Может использоваться только с ключом `-l`. С данным параметром в протокол выводится сводка по интервалам, как описано ниже.

`--log-prefix=`*`префикс`*

Задать префикс имён файлов для файлов протоколов, создаваемых с ключом `--log`. Префикс по умолчанию — `pgbench_log`.

`--progress-timestamp`

При отображении прогресса (с параметром `-P`) выводить текущее время (в формате Unix), а не количество секунд от начала запуска. Время задаётся в секундах с точностью до миллисекунд. Это помогает сравнивать журналы, записываемые разными средствами.

`--sampling-rate=`*`скорость передачи`*

Частота выборки для записи данных в протокол, изменяя которую можно уменьшить объём протокола. При указании этого параметра в протокол выводится информация только о заданном проценте транзакций. Со значением 1.0 в нём будут отмечаться все транзакции, а с 0.05 только 5%.

Обрабатывая протокол, не забудьте учесть частоту выборки. Например, вычисляя скорость (TPS), вам нужно будет соответственно умножить содержащиеся в нём числа (то есть, с частотой выборки 0.01 вы получите только 1/100 фактической скорости).

Общие параметры

pgbench принимает следующие общие аргументы командной строки:

`-h` *`компьютер`*
`--host=`*`компьютер`*

Адрес сервера баз данных

`-p` *`порт`*
`--port=`*`порт`*

Номер порта сервера баз данных

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Имя пользователя для подключения

`-V`
`--version`

Вывести версию pgbench и завершиться.

`-?`
`--help`

Вывести справку об аргументах командной строки pgbench и завершиться.

Замечания

Каково содержание «транзакции», которую выполняет pgbench?

Программа pgbench выполняет тестовые скрипты, выбирая их случайным образом из заданного списка. Это могут быть как встроенные скрипты, задаваемые аргументами `-b`, так и пользовательские, задаваемые аргументами `-f`. Для каждого скрипта можно задать относительный вес после `@`, чтобы скорректировать вероятность его выбора. По умолчанию вес считается равным `1`. Скрипты с весом `0` игнорируются.

Стандартный встроенный скрипт (также вызываемый с ключом `-b tpcb-like`) выдаёт семь команд в транзакции со случайно выбранными `aid`, `tid`, `bid` и `delta`. Его сценарий написан по мотивам теста производительности TPC-B, но это не собственно TPC-B, потому он называется так.

1. `BEGIN;`
2. `UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;`

3. `SELECT abalance FROM pgbench_accounts WHERE aid = :aid;`
4. `UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;`
5. `UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;`
6. `INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);`
7. `END;`

При выборе встроенного скрипта `simple-update` (или указании `-N`) шаги 4 и 5 исключаются из транзакции. Это позволяет избежать конкуренции при обращении к этим таблицам, но тест становится ещё менее похожим на TPC-B.

При выборе встроенного теста `select-only` (или указании `-S`) выполняется только `SELECT`.

Пользовательские скрипты

Программа pgbench поддерживает запуск пользовательских сценариев оценки производительности, позволяя заменять стандартный скрипт транзакции (описанный выше) скриптом, считываемым из файла (с параметром `-f`). В этом случае «транзакцией» считается одно выполнение данного скрипта.

Файл скрипта содержит одну или несколько команд SQL, разделённых точкой с запятой. Пустые строки и строки, начинающиеся с `--`, игнорируются. В файлах скриптов также могут содержаться «метакоманды», которые обрабатывает сама программа pgbench, как описано ниже.

> **Примечание**
>
> До версии Postgres Pro 9.6, SQL-команды в файлах скриптов завершались символами перевода строки, и поэтому они не могли занимать несколько строк. Теперь для разделения последовательных команд SQL *требуется* добавлять точку с запятой (хотя без неё можно обойтись в конце SQL-команды, за которой идёт метакоманда). Если вам нужно создать файл скрипта, работающий и со старыми версиями pgbench, записывайте каждую команду SQL в отдельной строке и завершайте её точкой с запятой.

Для файлов скриптов реализован простой механизм подстановки переменных. Переменные можно задать в командной строке параметрами `-D`, описанными выше, или метакомандами, рассматриваемыми ниже. Помимо переменных, которые можно установить параметрами командной строки `-D`, есть несколько автоматически устанавливаемых переменных; они перечислены в Таблице 244. Если значение этих переменных задаётся в параметре `-D`, оно переопределяет автоматическое значение. Когда значение переменной определено, его можно вставить в команду SQL, написав `:`*`имя_переменной`*. Каждый клиентский сеанс, если их несколько, получает собственный набор переменных.

Таблица 244. Автоматические переменные

Переменная	Описание
`scale`	текущий коэффициент масштаба
`client_id`	уникальное число, идентифицирующее клиентский сеанс (начиная с нуля)

Метакоманды в скрипте начинаются с обратной косой черты (`\`) и обычно продолжаются до конца строки, хотя их можно переносить на следующую строку последовательностью символов: обратная косая, возврат каретки. Аргументы метакоманд разделяются пробелами. Поддерживаемые метакоманды представлены ниже:

`\set` *`имя_переменной выражение`*

Устанавливает для переменной *`имя_переменной`* значение, вычисленное из *`выражения`*. Выражение может содержать целочисленные константы (например, `5432`), константы

с плавающей точкой двойной точности (например, `3.14159`), ссылки на переменные (`:имя_переменной`), унарные операторы (`+`, `-`), бинарные операторы (`+`, `-`, `*`, `/`, `%`), вычисляемые в обычном порядке, вызовы функций, а также скобки.

Примеры:

```
\set ntellers 10 * :scale
\set aid (1021 * random(1, 100000 * :scale)) % \
           (100000 * :scale) + 1
```

`\sleep` *номер* `[ us | ms | s ]`

Приостанавливает выполнение скрипта на заданное число микросекунд (`us`), миллисекунд (`ms`) или секунд (`s`). Когда единицы не указываются, подразумеваются секунды. Здесь *число* может быть целочисленной константой или ссылкой `:`*имя_переменной* на переменную с целочисленным значением.

Пример:

```
\sleep 10 ms
```

`\setshell` *имя_переменной команда* `[` *аргумент* `... ]`

Присваивает переменной *имя_переменной* результат команды оболочки *команда* с указанными *аргументами*. Эта команда должна просто выдать целочисленное значение в стандартный вывод.

Здесь *команда* и каждый *аргумент* может быть либо текстовой константой, либо ссылкой на переменную `:`*имя_переменной*. Если вы хотите записать *аргумент*, начинающийся с двоеточия, добавьте перед *аргументом* дополнительное двоеточие.

Пример:

```
\setshell назначаемая_переменная команда
 строковый_аргумент :переменная ::строка_начинающаяся_двоеточием
```

`\shell` *команда* `[` *аргумент* `... ]`

Действует так же, как и `\setshell`, но не учитывает результат команды.

Пример:

```
\shell команда строковый_аргумент :переменная ::строка_начинающаяся_двоеточием
```

Встроенные функции

Функции, перечисленные в Таблице 245, встроены в pgbench и могут применяться в выражениях в метакоманде `\set`.

Таблица 245. Функции pgbench

Функция	Тип результата	Описание	Пример	Результат
`abs( a)`	то же, что и *a*	модуль числа (абсолютное значение)	`abs(-17)`	`17`
`debug( a)`	то же, что и *a*	выводит *a* в stderr и возвращает *a*	`debug(5432.1)`	`5432.1`
`double( i)`	double	приведение к типу с плавающей точкой	`double(5432)`	`5432.0`
`greatest( a [, ... ] )`	double, если любой из аргументов (*a*)	наибольшее значение среди аргументов	`greatest(5, 4, 3, 2)`	`5`

Функция	Тип результата	Описание	Пример	Результат
	— double, а иначе целое число			
`int( x)`	integer	приведение к целочисленному типу	`int(5.4 + 3.8)`	`9`
`least( a [, ... ] )`	double, если любой из аргументов (*a*) — double, а иначе целое число	наименьшее значение среди аргументов	`least(5, 4, 3, 2.1)`	`2.1`
`pi()`	double	значение константы PI	`pi()`	`3.14159265358979323846`
`random( lb, ub)`	integer	случайное целое число с равномерным распределением в интервале `[lb, ub]`	`random(1, 10)`	целое между `1` и `10`
`random_exponential( lb, ub, parameter)`	integer	случайное целое число с экспоненциальным распределением в интервале `[lb, ub]` , см. ниже	`random_exponential(1, 10, 3.0)`	целое между `1` и `10`
`random_gaussian( lb, ub, parameter)`	integer	целое число с распределением Гаусса в интервале `[lb, ub]` , см. ниже	`random_gaussian(1, 10, 2.5)`	целое между `1` и `10`
`sqrt( x)`	double	квадратный корень	`sqrt(2.0)`	`1.414213562`

Функция `random` выдаёт значения с равномерным распределением, то есть вероятности получения всех чисел в интервале равны. Функции `random_exponential` и `random_gaussian` требуют указания дополнительного параметра типа double, определяющего точную форму распределения.

- Для экспоненциального распределения *`parameter`* управляет распределением, обрезая быстро спадающее экспоненциальное распределение в точке *`parameter`*, а затем это распределение проецируется на целые числа между границами. Точнее говоря, с

  f(x) = exp(-parameter * (x - min) / (max - min + 1)) / (1 - exp(-parameter))

  значение *`i`* между *`min`* и *`max`* выдаётся с вероятностью: `f(i) - f(i + 1)`.

  Интуиция подсказывает, что чем больше *`parameter`*, тем чаще будут выдаваться значения, близкие к *`min`*, и тем реже значения, близкие к *`max`*. Чем *`parameter`* ближе к 0, тем более плоским (более равномерным) будет распределение. В грубом приближении при таком распределении наиболее частый 1% значений в диапазоне рядом с *`min`* выдаётся *`parameter`*% времени. Значение *`parameter`* должно быть строго положительным.
- Для распределения Гаусса по интервалу строится обычное нормальное распределение (классическая кривая Гаусса в форме колокола) и этот интервал обрезается в точке `-parameter` слева и `+parameter` справа. Вероятнее всего при таком распределении выдаются значения из середины интервала. Точнее говоря, если `PHI(x)` — функция распределения нормальной случайной величины со средним значением `mu`, равным `(max + min) / 2.0`, и

  f(x) = PHI(2.0 * parameter * (x - mu) / (max - min + 1)) /

(2.0 * PHI(parameter) - 1)

тогда значение *i* между *min* и *max* включительно выдаётся с вероятностью: `f(i + 0.5) - f(i - 0.5)`. Интуиция подсказывает, что чем больше *parameter*, тем чаще будут выдаваться значения в середине интервала, и тем реже значения у границ *min* и *max*. Около 67% значений будут выдаваться из среднего интервала `1.0 / parameter`, то есть плюс/минус `0.5 / parameter` от среднего значения, и 95% из среднего интервала `2.0 / parameter`, то есть плюс/минус `1.0 / parameter` от среднего значения; например, если *parameter* равен 4.0, 67% значений выдаются из средней четверти (1.0 / 4.0) интервала (то есть от `3.0 / 8.0` до `5.0 / 8.0`) и 95% из средней половины (`2.0 / 4.0`) интервала (из второй и третьей четвертей). Чтобы преобразование Бокса-Мюллера было быстрым, *parameter* должен быть не меньше 2.0.

В качестве примера взгляните на встроенное определение транзакции типа TPC-B:

```
\set aid random(1, 100000 * :scale)
\set bid random(1, 1 * :scale)
\set tid random(1, 10 * :scale)
\set delta random(-5000, 5000)
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES
 (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
END;
```

С таким скриптом транзакция на каждой итерации будет обращаться к разным, случайно выбираемым строкам. (Этот пример показывает, почему важно, чтобы в каждом клиентском сеансе были собственные переменные — в противном случае они не будут независимо обращаться к разным строкам.)

Протоколирование транзакций

С параметром `-l` (но без `--aggregate-interval`), pgbench записывает информацию о каждой транзакции в протокол. Этот файл протокола будет называться *префикс*`.`*nnn*, где *префикс* по умолчанию — `pgbench_log`, а *nnn* — PID процесса pgbench. Префикс можно сменить, воспользовавшись ключом `--log-prefix`. Если параметр `-j` равен 2 или выше, будет создано несколько рабочих потоков, и каждый будет записывать отдельный протокол. Первый рабочий процесс будет использовать файл с тем же именем, что и в стандартном случае с одним потоком, а файлы остальных потоков будут называться *префикс*`.`*nnn*`.`*mmm*, где *mmm* — последовательный номер рабочего процесса, начиная с 1.

Протокол имеет следующий формат:

код_клиента число_транзакций длительность номер_скрипта время_эпохи время_мкс
 [*отставание_от_графика*]

Здесь *код_клиента* показывает, в сеансе какого клиента запускалась транзакция, *число_транзакций* отражает, сколько транзакций выполнялось в этом сеансе, *длительность* — общее время транзакций (в микросекундах), *номер_скрипта* показывает, какой файл скрипта использовался (это полезно при указании нескольких скриптов ключами `-f` и `-b`), а *время_эпохи*/*время_мкс* — отметка времени в формате Unix и смещение в микросекундах (из этих чисел можно получить время стандарта ISO 8601 с дробными секундами), показывающие, когда транзакция была завершена. Поле *отставание_от_графика* представляет разницу между запланированным временем запуска транзакции и фактическим временем запуска (в микросекундах). Оно выводится, только когда применяется параметр `--rate`. Когда одновременно применяются параметры `--rate` и `--latency-limit`, в поле *длительность* для пропущенных транзакций будет выводиться `skipped`.

Фрагмент протокола, полученного при выполнении с одним клиентом:

```
0 199 2241 0 1175850568 995598
0 200 2465 0 1175850568 998079
0 201 2513 0 1175850569 608
0 202 2038 0 1175850569 2663
```

Ещё один пример с `--rate=100` и `--latency-limit=5` (обратите внимание на дополнительный столбец *отставание_от_графика*):

```
0 81 4621 0 1412881037 912698 3005
0 82 6173 0 1412881037 914578 4304
0 83 skipped 0 1412881037 914578 5217
0 83 skipped 0 1412881037 914578 5099
0 83 4722 0 1412881037 916203 3108
0 84 4142 0 1412881037 918023 2333
0 85 2465 0 1412881037 919759 740
```

В этом примере транзакция 82 опоздала, так как её длительность (6.173 мс) превысила ограничение в 5 мс. Следующие две транзакции были пропущены, так как было слишком поздно их начинать.

Когда проводится длительное тестирование с большим количеством транзакций, файлы протоколов могут быть очень объёмными. Чтобы в них записывалась только случайная выборка транзакций, можно запустить команду с параметром `--sampling-rate`.

Протоколирование с агрегированием

С параметром `--aggregate-interval` протоколы имеют несколько иной формат:

начало_интервала число_транзакций сумма_длительности сумма_длительности_2 мин_длительность м
 [*сумма_задержки сумма_задержки_2 мин_задержка макс_задержка* [*пропущено_транзакций*]]

Здесь *начало_интервала* — начальное время интервала (в формате времени UNIX), *число_транзакций* — количество транзакций в данном интервале, *сумма_длительности* — суммарная длительность транзакций, *сумма_длительности_2* — сумма квадратов длительностей транзакций в интервале, *мин_длительность* — минимальная длительность в интервале, а *макс_задержка* — максимальная. Следующие поля, *сумма_задержки*, *сумма_задержки_2*, *мин_задержка* и *макс_задержка*, присутствуют, только если применяется параметр `--rate`. Они отражают время, на которое задержалась каждая транзакция, ожидая завершения предыдущей, то есть разницу между временем фактического запуска и запланированным временем. Самое последнее поле, *пропущено_транзакций*, тоже присутствует, только если применяется ключ `--latency-limit`. В нём выводится число транзакций, пропущенных из-за того, что их запуск задержался слишком надолго. Каждая транзакция учитывается в том интервале, в котором она была зафиксирована.

Пример вывода:

```
1345828501 5601 1542744 483552416 61 2573
1345828503 7884 1979812 565806736 60 1479
1345828505 7208 1979422 567277552 59 1391
1345828507 7685 1980268 569784714 60 1398
1345828509 7073 1979779 573489941 236 1411
```

Заметьте, что простой протокол (без агрегирования) показывает, какой скрипт использовался для каждой транзакции, в отличие от протокола с агрегированием. Таким образом, если вам нужны подобные сведения, но в разрезе скриптов, вам придётся агрегировать данные самостоятельно.

Время ожидания по операторам

С параметром `-r` программа pgbench учитывает время выполнения каждого оператора каждым клиентом. Затем по завершении тестирования она выводит среднее по всем значениям как время ожидания каждого оператора.

Со стандартным скриптом вывод будет примерно таким:

```
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
latency average = 15.844 ms
latency stddev = 2.715 ms
tps = 618.764555 (including connections establishing)
tps = 622.977698 (excluding connections establishing)
script statistics:
 - statement latencies in milliseconds:
         0.002  \set aid random(1, 100000 * :scale)
         0.005  \set bid random(1, 1 * :scale)
         0.002  \set tid random(1, 10 * :scale)
         0.001  \set delta random(-5000, 5000)
         0.326  BEGIN;
         0.603  UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid
 = :aid;
         0.454  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
         5.528  UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid
 = :tid;
         7.335  UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid
 = :bid;
         0.371  INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES
 (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
         1.212  END;
```

Если задействуется несколько файлов скриптов, средние значения выводятся отдельно для каждого файла.

Учтите, что сбор дополнительных временных показателей влечёт некоторые издержки и приводит к снижению средней скорости и, как результат, падению TPS. На сколько именно снизится скорость, во многом зависит от платформы и оборудования. Хороший способ оценить, каковы эти издержки — сравнить средние значения TPS, получаемые с подсчётом времени операторов и без такого подсчёта.

Полезные советы

Используя pgbench, можно без особого труда получить абсолютно бессмысленные числа. Последуйте приведённым советам, чтобы получить полезные результаты.

Во-первых, *никогда* не доверяйте тестам, которые выполняются всего несколько секунд. Воспользуйтесь параметром `-t` и `-T` и установите время выполнения не меньше нескольких минут, чтобы избавиться от шума в средних значениях. В некоторых случаях для получения воспроизводимых результатов тестирование должно продолжаться несколько часов. Чтобы понять, были ли получены воспроизводимые значения, имеет смысл запустить тестирование несколько раз.

Для стандартного сценария по типу TPC-B начальный коэффициент масштаба (`-s`) должен быть не меньше числа клиентов, с каким вы намерены проводить тестирование (`-c`); в противном случае вы, по большому счёту, будете замерять время конкурентных изменений. Таблица `pgbench_branches` содержит всего `-s` строк, а каждая транзакция хочет изменить одну из них, так что если значение `-c` превышает `-s`, это несомненно приведёт к тому, что многие транзакции будут блокироваться другими.

Стандартный сценарий тестирования также довольно сильно зависит от того, сколько времени прошло с момента инициализации таблиц: накопление неактуальных строк и «мёртвого»

пространства в таблицах влияет на результаты. Чтобы правильно оценить результаты, необходимо учитывать, сколько всего изменений было произведено и когда выполнялась очистка. Если же включена автоочистка, это может быть чревато непредсказуемыми изменениями оценок производительности.

Полезность результатов pgbench также может ограничиваться тем, что тестирование с большим числом клиентских сеансов само по себе нагружает систему. Этого можно избежать, запуская pgbench на другом компьютере, не на сервере баз данных, хотя при этом большое значение имеет скорость сети. Иногда, оценивая производительность одного сервера, полезно запускать даже несколько экземпляров pgbench параллельно, на отдельных клиентских компьютерах.

Безопасность

Если к базе данных, которая не приведена в соответствие шаблону безопасного использования схем, имеют доступ недоверенные пользователи, не запускайте pgbench в этой базе. Программа pgbench использует неполные имена и не настраивает для себя путь поиска.

pg_config

pg_config — вывести информацию об установленной версии Postgres Pro

Синтаксис

`pg_config` [*параметр*...]

Описание

Утилита pg_config выводит параметры конфигурации текущей установленной версии Postgres Pro. Это помогает, например, найти заголовочные файлы и библиотеки, требующиеся программным средствам, которые хотят взаимодействовать с Postgres Pro.

Параметры

При использовании pg_config можно передать следующие параметры:

`--bindir`

Вывести расположение исполняемых файлов. Можно использовать, например, для поиска утилиты `psql`. Обычно там же находится и сама утилита `pg_config`.

`--docdir`

Вывести расположение файлов документации.

`--htmldir`

Вывести расположение файлов документации в формате HTML.

`--includedir`

Вывести расположение заголовочных C-файлов клиентских интерфейсов.

`--pkgincludedir`

Вывести расположение других заголовочных C-файлов.

`--includedir-server`

Вывести расположение заголовочных C-файлов для программирования серверной части.

`--libdir`

Вывести расположение библиотек объектного кода.

`--pkglibdir`

Вывести расположение динамически подгружаемых модулей, либо путь, где сервер должен их искать. По этому пути также могут размещаться и другие архитектурно-зависимые файлы.

`--localedir`

Вывести расположение файлов поддержки локалей. Если поддержка локалей не была сконфигурирована на этапе сборки Postgres Pro, будет выведена пустая строка.

`--mandir`

Вывести расположение страниц руководства man.

`--sharedir`

Вывести расположение архитектурно-независимых вспомогательных файлов.

`--sysconfdir`

Вывести расположение системных конфигурационных файлов.

`--pgxs`

Вывести расположение файлов сборки расширений.

`--configure`

Вывести список параметров `configure`, использованных при сборке Postgres Pro. Это может пригодиться, чтобы при последующей сборке сделать идентичную конфигурацию. Или для того, чтобы найти с какими параметрами был собран используемый бинарный пакет. (Стоит отметить, что бинарные пакеты нередко содержат патчи, специфичные для дистрибутивов.) См. примеры ниже.

`--cc`

Вывести использованное при сборке Postgres Pro значение переменной `CC`. Оно отражает, какой C-компилятор применялся.

`--cppflags`

Вывести использованное при сборке Postgres Pro значение переменной `CPPFLAGS`. Оно отражает флаги C-компилятора, применённые для препроцессора. Обычно это флаги `-I`.

`--cflags`

Вывести использованное при сборке Postgres Pro значение переменной `CFLAGS`. Оно отражает флаги C-компилятора, применённые при сборке.

`--cflags_sl`

Вывести использованное при сборке Postgres Pro значение переменной `CFLAGS_SL`. Оно отражает дополнительные флаги C-компилятора для сборки разделяемых библиотек.

`--ldflags`

Вывести использованное при сборке Postgres Pro значение переменной `LDFLAGS`. Оно отражает флаги компоновщика.

`--ldflags_ex`

Вывести использованное при сборке Postgres Pro значение переменной `LDFLAGS_EX`. Оно отражает флаги компоновщика, использованные при сборке лишь исполняемых файлов.

`--ldflags_sl`

Вывести использованное при сборке Postgres Pro значение переменной `LDFLAGS_SL`. Оно отражает флаги компоновщика, использованные при сборке лишь разделяемых библиотек.

`--libs`

Вывести использованное при сборке Postgres Pro значение переменной `LIBS`. Обычно оно отражает флаги подключения внешних библиотек к Postgres Pro, переданные с ключом `-l`.

`--version`

Вывести версию PostgreSQL, на которой основан Postgres Pro.

`--pgpro-version`

Вывести версию Postgres Pro.

`--pgpro-edition`

Вывести редакцию Postgres Pro.

`-?`
`--help`

Вывести справку по команде pg_config.

Если одновременно передано несколько параметров, то выводимая информация будет следовать согласно их порядку. Если параметры не переданы, то будет выведена вся информация с подписями, к чему она относится.

Замечания

Параметры `--docdir`, `--pkgincludedir`, `--localedir`, `--mandir`, `--sharedir`, `--sysconfdir`, `--cc`, `--cppflags`, `--cflags`, `--cflags_sl`, `--ldflags`, `--ldflags_sl` и `--libs` доступны, начиная с версии PostgreSQL 8.1. Параметр `--htmldir` добавлен в PostgreSQL 8.4. Параметр `--ldflags_ex` добавлен в PostgreSQL 9.0.

Пример

Чтобы воспроизвести конфигурацию сборки текущей инсталляции Postgres Pro, можно выполнить команду:

```
eval ./configure `pg_config --configure`
```

Вывод `pg_config --configure` содержит символы экранирования, поэтому значения аргументов, содержащие пробелы, представлены корректно. Таким образом, для получения корректного результата необходимо применить `eval`.

pg_dump

pg_dump — выгрузить базу данных Postgres Pro в виде скрипта или в архивном формате

Синтаксис

`pg_dump` [*параметр-подключения*...] [*параметр*...] [*имя_бд*]

Описание

pg_dump — это программа для создания резервных копий базы данных Postgres Pro. Она создаёт целостные копии, даже если база параллельно используется. Программа pg_dump не препятствует доступу других пользователей к базе данных (ни для чтения, ни для записи).

Программа pg_dump выгружает только одну базу данных. Чтобы сохранить глобальные объекты, относящиеся ко всем базам в кластере, например, роли и табличные пространства, воспользуйтесь программой pg_dumpall.

Выгружаемые данные могут быть сохранены в виде скрипта, либо в одном из архивных форматов. Скрипты представляют собой текстовые файлы, содержащие SQL-команды, необходимые для воссоздания базы данных до состояния на момент создания скрипта. Для восстановления из скрипта его содержимое можно передать psql. Скрипты можно использовать для восстановления базы на других машинах, в том числе с иной архитектурой, а с некоторыми коррективами даже в других СУБД.

Для восстановления из архивных форматов файлов используется утилита pg_restore. Эти форматы позволяют указывать pg_restore какие объекты базы данных восстановить, а также позволяют изменить порядок следования восстанавливаемых объектов. Архивные форматы файлов спроектированы так, чтобы их можно были переносить на другие платформы с другой архитектурой.

Применение архивных форматов в сочетании утилит pg_restore и pg_dump позволяет организовывать эффективный механизм архивации и переноса данных. pg_dump можно использовать для резервирования всей базы данных, а затем при применении pg_restore выбрать нужные объекты для восстановления. Наиболее гибкие форматы выходных файлов это «custom» (`-Fc`) и «directory» (`-Fd`). Они позволяют выбрать и изменить порядок объектов, поддерживают восстановление в несколько потоков, а также сжимаются по умолчанию. При этом только формат «directory» поддерживает выгрузку данных в несколько потоков.

Во время работы pg_dump следует обращать внимание на предупреждения, которые печатаются в стандартный поток ошибок, особенно ввиду рассмотренных далее ограничений.

Параметры

Параметры командной строки для управления содержимым и форматом вывода.

имя_бд

Указывает имя базы данных, из которой будут выгружаться данные. Если имя не задано, то используется значение переменной окружения `PGDATABASE`. Если и переменная не задана, то в качестве имени базы будет взято имя пользователя, под которым осуществляется подключение.

`-a`
`--data-only`

Выводить только данные, но не схемы объектов (DDL). Будут копироваться данные таблиц, большие объекты, значения последовательностей.

Флаг похож на `--section=data`, но по историческим причинам не равнозначен ему.

`-b`
`--blobs`

Включить большие объекты в выгрузку. Это поведение по умолчанию при отсутствии ключей `--schema`, `--table` или `--schema-only`. Таким образом, ключ `-b` полезен, лишь когда нужно добавить большие объекты при выгрузке только избранной схемы или таблицы. Заметьте, что большие объекты относятся к данным, и поэтому будут выгружаться, когда используется ключ --data-only, но не ключ --schema-only.

`-B`
`--no-blobs`

Исключить из выгрузки большие объекты.

Когда задаётся и `-b`, и `-B`, большие объекты при выгрузке данных будут выводиться (см. описание ключа `-b`).

`-c`
`--clean`

Включить в выходной файл команды удаления (DROP) объектов базы данных перед командами создания (CREATE) этих объектов. Если дополнительно не указать флаг `--if-exists`, то при восстановлении в базу данных, где некоторые объекты отсутствуют, попытка удаления несуществующего объекта будет приводить к ошибке, которую можно игнорировать.

Этот параметр игнорируется, когда данные выгружаются в архивных форматах (не в текстовом). Для таких форматов данный параметр можно указать при вызове `pg_restore`.

`-C`
`--create`

Сформировать в начале вывода команду для создания базы данных и затем подключения к ней. В этом случае не важно, какая база указана в параметрах подключения перед выполнением скрипта. Также, если указан ключ `--clean`, то скрипт сначала удалит, а затем пересоздаст базу данных перед подключением к ней.

Этот параметр игнорируется, когда данные выгружаются в архивных форматах (не в текстовом). Для таких форматов данный параметр можно указать при вызове `pg_restore`.

`-E` *кодировка*
`--encoding=`*кодировка*

Создать копию в заданной кодировке. По умолчанию копия создаётся в кодировке, используемой базой данных. Другой способ достичь того же результата — задать желаемую кодировку в переменной окружения `PGCLIENTENCODING`.

`-f` *файл*
`--file=`*файл*

Отправить вывод в указанный файл. Параметр можно не указывать, если используется формат с выводом в файл. В этом случае будет использован стандартный вывод. Однако для формата с выводом в каталог параметр является обязательным и должен задавать путь к каталогу. В этом случае целевой каталог будет создан командой `pg_dump` и не должен существовать заранее.

`-F` *format*
`--format=`*format*

Указывает формат вывода копии. *format* может принимать следующие значения:

`p`
`plain`

Сформировать текстовый SQL-скрипт. Это поведение по умолчанию.

`c`
`custom`

Выгрузить данные в специальном архивном формате, пригодном для дальнейшего использования утилитой pg_restore. Наряду с форматом `directory` является наиболее гибким форматом, позволяющим вручную выбирать и сортировать восстанавливаемые объекты. Вывод в этом формате по умолчанию сжимается.

`d`
`directory`

Выгрузить данные в формате каталога. Этот формат пригоден для дальнейшего использования утилитой pg_restore. При этом будет создан каталог, в котором для каждой таблицы и большого объекта будут созданы отдельные файлы, а также файл оглавления в машинно-читаемом формате, понятном для pg_restore. С полученной резервной копией можно работать штатными средствами Unix, например, несжатую копию можно сжать посредством gzip. Этот формат по умолчанию сжимается, а также поддерживает работу в несколько потоков.

`t`
`tar`

Выгрузить данные в формате `tar`, для дальнейшего использования с утилитой pg_restore. Этот формат совместим с форматом вывода в каталог: если архив распаковать, получится корректная копия в формате каталога. Однако формат tar не поддерживает сжатие. Также, применяя формат tar, при восстановлении нельзя изменить относительный порядок элементов данных.

`-j` *`число_заданий`*
`--jobs=`*`число_заданий`*

Осуществить выгрузку в параллельном режиме, обрабатывая одновременно несколько таблиц (в количестве *`число_заданий`*). Это уменьшает время работы, но увеличивает нагрузку на сервер. Этот параметр можно использовать только с форматом вывода в каталог, так как это единственный формат, позволяющий нескольким процессам записывать данные одновременно.

pg_dump откроет *`число_заданий`* + 1 соединений с базой данных. Таким образом необходимо обеспечить достаточное значение параметра max_connections.

Если во время выгрузки в несколько потоков, параллельно работающие сессии будут запрашивать эксклюзивные блокировки на объекты базы данных, то pg_dump может завершиться аварийно. Дело в том, что головной процесс pg_dump вначале запрашивает разделяемые блокировки на объекты, которые позже будут выгружать рабочие процессы. Это делается для того, чтобы никто не смог удалить объекты на время работы pg_dump. Если же другая сессия запросит эксклюзивную блокировку на объект, то запрос на блокировку будет поставлен в очередь, до тех пор пока разделяемая блокировка головного процесса pg_dump не будет снята. В последующем, любая попытка доступа к этому объекту будет вставать в очередь, вслед за эксклюзивной блокировкой. В том числе в очередь попадет и рабочий процесс pg_dump. Если не принять меры предосторожности, то получим классическую взаимоблокировку. Для предупреждения подобных конфликтов, рабочий процесс pg_dump ещё раз запрашивает разделяемую блокировку на объект с указанием `NOWAIT`. И если он не смог получить блокировку, значит кто-то ещё запросил эксклюзивную блокировку объекта. А это значит, что нет возможности продолжить выгрузку, поэтому pg_dump прерывает дальнейшую работу.

Для получения целостной резервной копии сервер баз данных должен поддерживать функциональность синхронизированных снимков, которая была введена в версии PostgreSQL 9.2 для ведущих серверов и в 10 для ведомых. Это позволяет разным клиентам работать с одной и той же версией данных, несмотря на использование разных подключений. `pg_dump -j` использует множественные подключения. Первое подключение осуществляется головным

процессом, а последующие — рабочими процессами. Без функциональности синхронизируемых снимков нет гарантии того, что каждое подключение увидит одни и те же данные, что может привести к несогласованности данных резервной копии.

Если необходимо выполнить выгрузку в несколько потоков на сервере версии до 9.2, необходимо быть уверенным, что база данных не будет изменяться с момента подключения головного процесса и до момента, когда последний рабочий процесс подключится к базе данных. Для этого, проще всего перед запуском pg_dump остановить все процессы, модифицирующие данные (DML и DDL). Также, при запуске `pg_dump -j` на сервере PostgreSQL до версии 9.2 нужно указывать параметр `--no-synchronized-snapshots`.

`-n` *схема*
`--schema=`*схема*

Выгрузить только схемы, соответствующие шаблону *схема*; вместе с этими схемами будут выгружены и все содержащиеся в них объекты. Когда этот параметр отсутствует, выгружаются все несистемные схемы в целевой базе данных. Чтобы выгрузить несколько схем, ключ `-n` можно указать несколько раз. Кроме того, параметр *схема* интерпретируется по тем же правилам, что и шаблон в командах psql `\d` (см. Подраздел «Шаблоны поиска»), так что несколько схем можно выбрать и шаблоном со знаками подстановки. Используя знаки подстановки, при необходимости заключайте шаблон в кавычки, чтобы эти знаки не разворачивала оболочка системы; см. Раздел «Примеры».

> **Примечание**
>
> При использовании `-n`, pg_dump не выгружает объекты других схем, от которых выгружаемая схема может зависеть. Таким образом не гарантируется, что выгруженная схема будет успешно восстановлена в чистой базе данных.

> **Примечание**
>
> Не принадлежащие схемам объекты (например, большие бинарные объекты), не выгружаются с параметром `-n`. Однако можно указать `--blobs`, чтобы они попали в выгрузку.

`-N` *схема*
`--exclude-schema=`*схема*

Не выгружать схемы, соответствующие шаблону *схема*. Шаблон интерпретируется по тем же правилам, что и для параметра `-n`. Параметр `-N` можно использовать в команде несколько раз для исключения схем, соответствующих нескольким шаблонам.

При одновременном использовании параметров `-n` и `-N` будут выгружаться схемы, соответствующие шаблону параметра `-n` и не противоречащие шаблону параметра `-N`.

`-o`
`--oids`

Выгружать идентификаторы объектов (OIDs) вместе с данными таблиц. Используйте этот параметр, если в приложении есть ссылки на OID, например во внешних ключах. В противном случае этот параметр лучше не использовать.

`-O`
`--no-owner`

Не формировать команды, устанавливающие владельца объектов базы данных. По умолчанию pg_dump генерирует команды `ALTER OWNER` или `SET SESSION AUTHORIZATION` для назначения

владельцев объектов базы. Эти команды завершатся неудачно, если скрипт будет запущен не суперпользователем или не владельцем объектов. Чтобы создать скрипт, который можно выполнить при восстановлении от лица произвольного пользователя и назначить его в качестве владельца объектов восстанавливаемой базы, необходимо указать флаг `-O`.

Этот параметр игнорируется, когда данные выгружаются в архивных форматах (не в текстовом). Для таких форматов данный параметр можно указать при вызове `pg_restore`.

`-R`
`--no-reconnect`

Параметр является устаревшим, но в целях совместимости ещё работает.

`-s`
`--schema-only`

Выгружать только определения объектов (схемы), без данных.

Действие параметра противоположно действию `--data-only`. Это похоже на указание `--section=pre-data --section=post-data`, но по историческим причинам не равнозначно ему.

(Не путайте этот параметр с `--schema`, где слово «схема» используется в другом значении.)

Чтобы не выгружать данные отдельных таблиц, используйте параметр `--exclude-table-data`.

`-S` *имя_пользователя*
`--superuser=`*имя_пользователя*

Указать суперпользователя, который будет использоваться для отключения триггеров. Параметр имеет значение только вместе с `--disable-triggers`. Обычно его лучше не использовать, а запускать полученный скрипт от имени суперпользователя.

`-t` *таблица*
`--table=`*таблица*

Выгрузить только таблицы, соответствующие шаблону *таблица*. В этом контексте под «таблицей» подразумеваются также представления, материализованные представления, последовательности и сторонние таблицы. Чтобы выбрать несколько таблиц, ключ `-t` можно указать несколько раз. Кроме того, параметр *таблица* интерпретируется по тем же правилам, что и шаблон в командах psql `\d` (см. Подраздел «Шаблоны поиска»), так что несколько таблиц можно выбрать и с шаблоном со знаками подстановки. Используя знаки подстановки, при необходимости заключайте шаблон в кавычки, чтобы эти знаки не разворачивала оболочка системы; см. Раздел «Примеры».

Параметры `-n` и `-N` не действуют в присутствии параметра `-t`, так как отобранные им таблицы всё равно будут выгружены, а не табличные объекты выгружаться не будут.

> **Примечание**
>
> При использовании `-t`, pg_dump не выгружает прочие объекты, от которых выгружаемые таблицы могут зависеть. Таким образом не гарантируется, что выгруженные таблицы будут успешно восстановлены в чистой базе данных.

> **Примечание**
>
> Поведение параметра `-t` для версий ниже чем PostgreSQL 8.2 отличается от более поздних. Прежде, указание `-t таблица` включало все таблицы, соответствующие шаблону `таблица`, а сейчас это приведёт к выгрузке только тех таблиц, которые будут обнаружены

в текущем пути поиска. Для получения старого поведения можно использовать конструкцию вида `-t '*.таблица'`. Также, чтобы указать таблицу из конкретной схемы, сейчас лучше использовать `-t схема.таблица`, вместо старой конструкции `-n схема -t таблица`.

`-T` *таблица*
`--exclude-table=`*таблица*

Не выгружать таблицы, соответствующие шаблону *таблица*. Шаблон интерпретируется по тем же правилам, что и для параметра `-t`. Параметр `-T` можно использовать в команде несколько раз для исключения таблиц, соответствующих нескольким шаблонам.

При одновременном использовании параметров `-t` и `-T` будут выгружаться таблицы, соответствующие шаблону параметра `-t` и не противоречащие шаблону параметра `-T`.

`-v`
`--verbose`

Включить подробный режим. pg_dump будет выводить в стандартный поток ошибок подробные комментарии к объектам, включая время начала и окончания выгрузки, а также сообщения о прогрессе выполнения.

`-V`
`--version`

Вывести версию pg_dump.

`-x`
`--no-privileges`
`--no-acl`

Не выгружать права доступа (команды GRANT/REVOKE).

`-Z` *0..9*
`--compress=`*0..9*

Установить уровень сжатия данных. Ноль означает, что сжатие выключено. Для специального формата и формата каталога будут сжиматься файлы отдельных таблиц. По умолчанию применяется умеренный уровень сжатия. Если указать отличный от нулевого уровень сжатия для простого формата, то сжиматься будет весь выходной файл, как это было бы при передаче файла команде gzip. Однако по умолчанию для простого формата сжатие не производится. Формат `tar` в настоящий момент не поддерживает сжатие.

`--add-collprovider`

Если провайдер правил сортировки для выгружаемой базы данных задаётся неявно, этот параметр добавляет явное указание провайдера в выгрузку. Этот вариант рекомендуется, если вы используете pg_dump и pg_restore для обновления с предыдущих версий Postgres Pro или любой версии PostgreSQL. Это позволяет сохранить изначальный провайдер основного правила сортировки для переносимой базы данных, если в базе `template0` в новом кластере используются другие параметры правил сортировки. В противном случае ограничения-проверки, которые используют основное правило сортировки в базе данных, могут поменять поведение, и команда `COPY` завершится ошибкой. При обновлении в двоичном формате провайдер сохраняется автоматически.

`--binary-upgrade`

Этот параметр предназначен для утилит обновления сервера. Использование для иных целей не рекомендуется и не поддерживается. Поведение параметра может быть изменено в последующих версиях без предварительного уведомления.

--column-inserts
--attribute-inserts

Выгружать данные таблиц в виде команд INSERT с явным указанием столбцов (INSERT INTO *таблица* (*столбец*, ...) VALUES ...). Скорость восстановления при этом значительно снизится, но данный вариант оправдан, когда загружать данные нужно не в Postgres Pro. Также, поскольку для каждой строки генерируется отдельная команда, сбой при последующей загрузке приведёт к потере конкретной строки, а не всей таблицы.

--disable-dollar-quoting

Этот параметр запрещает заключать в доллары тело функций, что оставляет возможность только заключать их в кавычки, применяя стандартный синтаксис SQL.

--disable-triggers

Используется при выгрузке одних данных. Указывает pg_dump включать в вывод команды для временного выключения триггеров при восстановлении в целевой базе данных. Применяется в ситуациях, когда существуют проверки ссылочной целостности или другие триггеры, которые необходимо выключить на время восстановления.

В настоящее время команды, генерируемые с параметром --disable-triggers, должны исполняться от имени суперпользователя. Таким образом, необходимо также передавать флаг -S, либо при восстановлении выполнять скрипт от имени суперпользователя.

Этот параметр игнорируется, когда данные выгружаются в архивных форматах (не в текстовом). Для таких форматов данный параметр можно указать при вызове pg_restore.

--enable-row-security

Этот параметр имеет смысл только при выгрузке содержимого таблицы, для которой включена защита строк. По умолчанию pg_dump устанавливает для row_security значение off, чтобы убедиться, что выгружаются все данные из таблицы. Если пользователь не имеет достаточных прав для обхода защиты строк, выдаётся ошибка. Этот параметр указывает pg_dump включить row_security, что позволит пользователю выгрузить часть содержимого таблицы, к которой он имеет доступ.

Заметьте, что в настоящее время для использования этого параметра обычно желательно, чтобы данные были выгружены в формате INSERT, так как команда COPY FROM в процессе восстановления не поддерживает защиту строк.

--exclude-table-data=*таблица*

Не выгружать содержимое таблиц, соответствующих шаблону *таблица*. Шаблон таблицы интерпретируется по тем же правилам, что и для параметра -t. Параметр --exclude-table-data можно использовать в команде несколько раз для исключения таблиц, соответствующих нескольким шаблонам. Полезно, когда нужно получить определение таблицы, без содержимого.

Чтобы не выгружать содержимое всех таблиц базы, используйте параметр --schema-only.

--if-exists

При очистке целевой базы использовать условные команды (добавлять предложение IF EXISTS). Применяется только с параметром --clean.

--inserts

Выгружать данные таблиц в виде команд INSERT вместо COPY. Скорость восстановления при этом значительно снизится, но данный вариант оправдан, когда загружать данные нужно не в Postgres Pro. Также, поскольку для каждой строки генерируется отдельная команда, сбой при последующей загрузке приведёт к потере конкретной строки, а не всей таблицы. Заметьте, что восстановление может провалиться полностью, если у таблицы изменён порядок столбцов.

В такой ситуации можно использовать параметр `--column-inserts`, для которого порядок столбцов не важен, но он работает ещё медленнее.

`--lock-wait-timeout=`*`время_ожидания`*

Не ждать бесконечно получения разделяемых блокировок таблиц в начале процедуры выгрузки. Вместо этого выдать ошибку, если не удастся заблокировать таблицы за указанное *`время_ожидания`*. Это время можно задать в любом из форматов, принимаемых командой `SET statement_timeout`. (Допустимые форматы зависят от версии сервера, выгружающего данные, но количество миллисекунд в виде целого числа принимают все версии.)

`--no-publications`

Не выгружать публикации.

`--no-security-labels`

Не выгружать метки безопасности.

`--no-subscriptions`

Не выгружать подписки.

`--no-sync`

По умолчанию `pg_dump` ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром `pg_dump` завершается немедленно, то есть выполняется быстрее, но в случае неожиданного сбоя операционной системы выгруженные данные могут оказаться испорченными. Вообще этот параметр предназначен прежде всего для тестирования, для производственной среды он не подходит.

`--no-synchronized-snapshots`

Позволяет запускать `pg_dump -j` на серверах с версией ниже чем 9.2. Подробнее в описании параметра `-j`.

`--no-tablespaces`

Не формировать команды для указания табличных пространств. При восстановлении все объекты будут создаваться в табличном пространстве по умолчанию.

Этот параметр игнорируется, когда данные выгружаются в архивных форматах (не в текстовом). Для таких форматов данный параметр можно указать при вызове `pg_restore`.

`--no-unlogged-table-data`

Не выгружать данные нежурналируемых таблиц. Параметр не влияет на выгрузку определений таблиц, он только подавляет вывод содержимого таблиц. С резервного сервера содержимое нежурналируемых таблиц не выгружается никогда.

`--quote-all-identifiers`

Принудительно экранировать все идентификаторы. Этот параметр рекомендуется при выгрузке базы, когда основная версия сервера PostgreSQL, с которого выгружается база, отличается от версии pg_dump, или когда выгруженная копия предназначена для загрузки на сервере с другой основной версией. По умолчанию pg_dump экранирует только те идентификаторы, которые являются зарезервированными словами в собственной основной версии. Иногда это приводит к проблемам совместимости с серверами других версий, в которых множество зарезервированных слов может быть несколько другим. Применение параметра `--quote-all-identifiers` предотвращает подобные проблемы, ценой ухудшения читаемости скрипта с выгруженными данными.

`--section=`*`имя_секции`*

Выгружать лишь указанную секцию. Имя секции может принимать значения `pre-data`, `data` или `post-data`. Для выгрузки нескольких секций, параметр можно использовать несколько раз в одной команде. По умолчанию резервируются все секции.

Секция data содержит непосредственно данные таблиц, больших объектов и значения последовательностей. Секция post-data содержит определения индексов, триггеров, правил и ограничений (кроме ограничений проверки, созданных без NOT VALID). Секция pre-data включает определения остальных элементов.

--serializable-deferrable

Использовать при выгрузке транзакцию с уровнем изоляции serializable для получения снимка, согласованного с последующими состояниями базы. Правда для этого нужно выждать момент, когда в потоке транзакций нет аномалий, и поэтому нет риска, что выгрузка завершится неудачно, и риска отката других транзакций с ошибкой serialization_failure. Более подробно изоляция транзакций и управление одновременным доступом описывается в Главе 13.

Параметр не особо полезен в случаях, когда требуется восстановление после сбоя. Он полезен для создания копии базы данных, в которой формируются отчёты и выполняются другие операции чтения, в то время как в основной базе продолжается обычная работа. Без этого параметра выгрузка может содержать не целостное состояние базы данных. Например, если используется пакетная обработка, статус пакета может отражаться как завершённый, в то время как в выгрузке будут не все элементы пакета.

Параметр не будет влиять на результат, если во время запуска pg_dump нет активных транзакций на чтение-запись. Если же активные транзакции чтения-записи есть, то начало выгрузки может быть отложено на неопределённый период времени. После того как выгрузка началась, производительность с этим ключом или без него будет одинаковой.

--snapshot=*имя_снимка*

Использовать заданный синхронный снимок при выгрузке данных из базы (за подробностями обратитесь к Таблице 9.82).

Этот параметр полезен, когда требуется синхронизировать выгружаемые данные со слотом логической репликации (см. Главу 46) или с другим одновременным сеансом.

В случае с параллельной выгрузкой будет использоваться имя снимка, определённое этим параметром; новый снимок не будет сделан.

--strict-names

Требует, чтобы каждому указанию схемы (-n/--schema) и таблицы (-t/--table) соответствовала минимум одна схема/таблица в выгружаемой базе данных. Заметьте, что если не находится вообще ни одной схемы/таблицы для заданных шаблонов, pg_dump выдаёт ошибку и без ключа --strict-names.

Этот параметр не действует на ключи -N/--exclude-schema, -T/--exclude-table или --exclude-table-data. Если не находятся объекты, соответствующие шаблонам исключения, это не считается ошибкой.

--use-set-session-authorization

Выводить команды SET SESSION AUTHORIZATION, соответствующие стандарту, вместо ALTER OWNER, для назначения владельцев объектов. В результате выгруженный скрипт будет более стандартизированным, но может не восстановиться корректно, в зависимости от истории объектов. Кроме того, для использования SET SESSION AUTHORIZATION при восстановлении нужны права суперпользователя, в то время как ALTER OWNER требует меньших привилегий.

-?
--help

Показать справку по аргументам командной строки pg_dump и завершиться.

Далее описаны параметры управления подключением.

`-d` *имя_бд*
`--dbname=`*имя_бд*

Указывает имя базы данных для подключения. Равнозначно указанию *имя_бд* в первом аргументе, не являющемся ключом, в командной строке. Вместо имени может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

`-h` *сервер*
`--host=`*сервер*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета. Значение по умолчанию берётся из переменной окружения `PGHOST`, если она установлена. В противном случае выполняется подключение к Unix-сокету.

`-p` *порт*
`--port=`*порт*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной окружения `PGPORT`, если она установлена, либо числом, заданным при компиляции.

`-U` *имя_пользователя*
`--username=`*имя_пользователя*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как pg_dump запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, pg_dump лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

`--role=`*имя роли*

Задаёт имя роли, которая будет осуществлять выгрузку. Получив это имя, pg_dump выполнит `SET ROLE` *имя_роли* после подключения к базе данных. Это полезно, когда проходящий проверку пользователь (указанный в `-U`) не имеет прав, необходимых для pg_dump, но может переключиться на роль, наделённую этими правами. В некоторых окружениях правила запрещают подключаться к серверу непосредственно суперпользователю, и этот параметр позволяет выполнить выгрузку, не нарушая их.

Переменные окружения

`PGDATABASE`
`PGHOST`
`PGOPTIONS`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию.

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 31.14).

Диагностика

pg_dump на низком уровне выполняет команды SELECT. Если есть проблемы с работой pg_dump, убедитесь, что в базе данных можно выполнить SELECT, например из psql. Также следует учитывать, что при этом применяются все свойства подключения по умолчанию и переменные окружения, которые использует клиентская библиотека libpq.

Обычно действия pg_dump в базе данных отслеживаются сборщиком статистики. Если это нежелательно, то можно установить параметр track_counts в false в переменной окружения PGOPTIONS или в команде ALTER USER.

Замечания

Если в базу данных кластера template1 устанавливались дополнительные объекты, то следует убедиться, что выгрузка pg_dump загружается в пустую базу данных. Иначе существует вероятность возникновения ошибок дублирования создаваемых объектов. Чтобы создать пустую базу данных, копируйте её из шаблона template0, вместо template1, например:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

Если выгружаются только данные с одновременным использованием --disable-triggers, pg_dump сформирует команды для выключения табличных триггеров перед вставкой данных, а после них — команды, включающие триггеры обратно. Если восстановление будет прервано в середине процесса, системный каталог может остаться в неверном состоянии.

Сформированный pg_dump файл не содержит данных статистики, используемых планировщиком. Поэтому после восстановления следует выполнить ANALYZE для достижения оптимальной производительности; за дополнительной информацией обратитесь к Подразделу 23.1.3 и Подразделу 23.1.6. Также не выгружаются команды ALTER DATABASE ... SET. Эти свойства, вместе с данными о ролях и прочими глобальными объектами, выгружаются с помощью pg_dumpall.

Так как pg_dump применяется для переноса данных в новые версии Postgres Pro, предполагается, что вывод pg_dump можно загрузить на сервер Postgres Pro более новой версии, чем версия pg_dump. pg_dump может также выгружать данные серверов Postgres Pro более старых версий, чем его собственная. (В настоящее время поддерживаются версии, начиная с PostgreSQL 8.0.) Однако утилита pg_dump не может выгружать данные с серверов Postgres Pro более новых основных версий; она не будет даже пытаться делать это, во избежание некорректной выгрузки. Также не гарантируется, что вывод pg_dump может быть загружен на сервере более старой основной версии — даже если данные были выгружены с сервера той же версии. Для загрузки такого файла на старом сервере может потребоваться вручную исправить в нём синтаксис, не воспринимаемый старой версией. Имея дело с разными версиями, рекомендуется применять параметр --quote-all-identifiers, так как он может предотвратить проблемы, возникающие при изменении множества зарезервированных слов в разных версиях PostgreSQL.

Выгружая подписки на логическую репликацию, pg_dump будет выдавать команды CREATE SUBSCRIPTION с указанием connect = false, так что при восстановлении подписки не будут устанавливаться удалённые подключения для создания слота репликации или для начального копирования таблиц. Таким образом, выгруженные данные могут быть восстановлены без сетевого доступа к удалённым серверам. Вновь активировать подписки должным образом — задача пользователя. Если задействованные серверы поменялись, возможно, придётся скорректировать строки подключения. Также может быть уместно опустошить целевые таблицы перед началом полного копирования таблиц.

Примеры

Выгрузка базы данных mydb в файл SQL-скрипта:

```
$ pg_dump mydb > db.sql
```

Восстановление из ранее полученного скрипта в чистую базу `newdb`:

```
$ psql -d newdb -f db.sql
```

Выгрузка базы данных в специальном формате:

```
$ pg_dump -Fc mydb > db.dump
```

Выгрузка базы данных в формате каталога:

```
$ pg_dump -Fd mydb -f dumpdir
```

Выгрузка базы данных в формате каталога в 5 параллельных потоков:

```
$ pg_dump -Fd mydb -j 5 -f dumpdir
```

Восстановление из архива в чистую новую базу данных `newdb`:

```
$ pg_restore -d newdb db.dump
```

Выгрузка отдельной таблицы `mytab`:

```
$ pg_dump -t mytab mydb > db.sql
```

Выгрузка всех таблиц, имена которых начинаются с `emp` и которые принадлежат схеме `detroit`, кроме таблицы `employee_log`:

```
$ pg_dump -t 'detroit.emp*' -T detroit.employee_log mydb > db.sql
```

Выгрузка всех схем, имена которых начинаются с `east` или `west`, заканчиваются на `gsm` и не содержат `test`:

```
$ pg_dump -n 'east*gsm' -n 'west*gsm' -N '*test*' mydb > db.sql
```

То же самое, но с использованием регулярного выражения:

```
$ pg_dump -n '(east|west)*gsm' -N '*test*' mydb > db.sql
```

Выгрузка всех объектов базы данных, кроме таблиц, имена которых начинаются с `ts_`:

```
$ pg_dump -T 'ts_*' mydb > db.sql
```

Чтобы указать имя в верхнем или смешанном регистре в ключе `-t` и связанных с ним, это имя нужно заключить в кавычки; иначе оно будет приведено к нижнему регистру (см. Подраздел «Шаблоны поиска»). Но кавычки являются спецсимволом для оболочки, поэтому и они, в свою очередь, должны заключаться в кавычки. Так, чтобы выгрузить одну таблицу с именем в смешанном регистре, нужно написать примерно следующее:

```
$ pg_dump -t "\"MixedCaseName\"" mydb > mytab.sql
```

См. также

pg_dumpall, pg_restore, psql

pg_dumpall

pg_dumpall — выгрузить кластер баз данных Postgres Pro в файл скрипта

Синтаксис

`pg_dumpall` [*параметр-подключения*...] [*параметр*...]

Описание

Утилита pg_dumpall предназначена для записи («выгрузки») всех баз данных кластера Postgres Pro в один файл в формате скрипта. Этот файл содержит команды SQL, так что передав его на вход psql, можно восстановить все базы данных. Чтобы его сформировать, pg_dumpall вызывает для каждой базы данных в кластере pg_dump и дополнительно выгружает глобальные объекты, общие для всех баз данных. (Утилита pg_dump не сохраняет эти объекты.) В настоящее время, к таким объектам относятся группы, пользователи и табличные пространства, а также такие свойства, как права доступа, которые применяются к базам данных в целом.

Так как утилита pg_dumpall читает таблицы из всех баз данных, для получения полного содержимого баз запускать её, как правило, нужно от имени суперпользователя. Чтобы впоследствии выполнить сохранённый скрипт, также необходимо иметь права суперпользователя, позволяющие создавать пользователей, группы и собственно базы данных.

Генерируемый SQL-скрипт записывается в стандартное устройство вывода. Чтобы перенаправить его в файл, воспользуйтесь параметром `-f`/`--file` или операторами оболочки.

Утилите pg_dumpall требуется подключаться к серверу Postgres Pro несколько раз (к каждой базе по отдельности). Если вы проходите проверку подлинности по паролю, вам придётся каждый раз вводить пароль. Чтобы избежать этого, удобно иметь файл `~/.pgpass`. За дополнительными сведениями обратитесь к Разделу 31.15.

Параметры

Параметры командной строки для управления содержимым и форматом вывода.

`-a`
`--data-only`

Выгружать только данные, без схемы (определений данных).

`-c`
`--clean`

Добавить команды SQL для удаления (DROP) баз данных перед командами, создающими их. В дополнение к ним добавляются команды `DROP` для ролей и табличных пространств.

`-f` *имя_файла*
`--file=`*имя_файла*

Направить вывод в указанный файл. Если этот параметр опущен, скрипт записывается в стандартный вывод.

`-g`
`--globals-only`

Выгружать только глобальные объекты (роли и табличные пространства), без баз данных.

`-o`
`--oids`

Выгружать идентификаторы объектов (OIDs) вместе с данными таблиц. Используйте этот параметр, если в приложении есть ссылки на OID, например во внешних ключах. В противном случае, этот параметр лучше не использовать.

`-O`
`--no-owner`

Не генерировать команды, устанавливающие владение объектами, как в исходной базе данных. По умолчанию, pg_dumpall генерирует команды `ALTER OWNER` или `SET SESSION AUTHORIZATION`, восстанавливающие исходных владельцев для создаваемых элементов схемы. Однако выполнить эти команды сможет только суперпользователь (или пользователь, владеющий всеми объектами, создаваемыми скриптом). Чтобы получить скрипт, который сможет восстановить любой пользователь (но при этом он станет владельцем всех объектов), используется `-O`.

`-r`
`--roles-only`

Выгружать только роли, без баз данных и табличных пространств.

`-s`
`--schema-only`

Выгружать только определения объектов (схемы), без данных.

`-S` *имя_пользователя*
`--superuser=`*имя_пользователя*

Указать суперпользователя, который будет использоваться для отключения триггеров. Параметр имеет значение только вместе с `--disable-triggers`. Обычно его лучше не использовать, а запускать полученный скрипт от имени суперпользователя.

`-t`
`--tablespaces-only`

Выгружать только табличные пространства, без баз данных и ролей.

`-v`
`--verbose`

Включить режим подробных сообщений. В этом режиме pg_dumpall записывает в выходной файл время начала/завершения выгрузки, а в стандартный канал ошибок — сообщения о процессе. При этом подробные сообщения будет также выводить pg_dump.

`-V`
`--version`

Сообщить версию pg_dumpall и завершиться.

`-x`
`--no-privileges`
`--no-acl`

Не выгружать права доступа (команды GRANT/REVOKE).

`--add-collprovider`

Если провайдер правил сортировки для выгружаемой базы данных задаётся неявно, этот параметр добавляет явное указание провайдера в выгрузку. Этот вариант рекомендуется, если вы используете pg_dumpall и pg_restore для обновления с предыдущих версий Postgres Pro или любой версии PostgreSQL. Это позволяет сохранить изначальный провайдер основного правила сортировки для переносимой базы данных, если в базе `template0` в новом кластере используются другие параметры правил сортировки. В противном случае ограничения-проверки, которые используют основное правило сортировки в базе данных, могут поменять поведение, и команда `COPY` завершится ошибкой. При обновлении в двоичном формате провайдер сохраняется автоматически.

`--binary-upgrade`

Этот параметр предназначен для утилит обновления сервера. Использование для иных целей не рекомендуется и не поддерживается. Поведение параметра может быть изменено в последующих версиях без предварительного уведомления.

`--column-inserts`
`--attribute-inserts`

Выгружать данные в виде команд `INSERT` с явно задаваемыми именами столбцов (`INSERT INTO` *`таблица`* `(`*`столбец`*`, ...) VALUES ...`). При этом восстановление будет очень медленным; в основном это применяется для выгрузки данных, которые затем будут загружаться не в Postgres Pro.

`--disable-dollar-quoting`

Этот параметр запрещает заключать в доллары тело функций, что оставляет возможность только заключать их в кавычки, применяя стандартный синтаксис SQL.

`--disable-triggers`

Этот параметр действует только при выгрузке одних данных. С ним pg_dumpall добавляет команды, отключающие триггеры в целевых таблицах на время загрузки данных. Используйте его, если в ваших таблицах определены проверки ссылочной целостности или другие триггеры, которые вы не хотели бы выполнять в процессе загрузки данных.

В настоящее время команды, генерируемые с параметром `--disable-triggers`, должны исполняться от имени суперпользователя. Таким образом, необходимо также передавать флаг `-S`, либо при восстановлении выполнять скрипт от имени суперпользователя.

`--if-exists`

Использовать условные команды (т. е. добавлять предложение `IF EXISTS`) при очистке базы данных и других объектов. Этот параметр принимается, только если также указан параметр `--clean`.

`--inserts`

Выгружать данные в виде команд `INSERT`, а не `COPY`. При этом восстановление значительно замедлится; в основном это применяется для выгрузки данных, которые затем будут загружаться не в Postgres Pro. Заметьте, что восстановление может вовсе не выполниться при изменении порядка столбцов в таблицах. В этом смысле параметр `--column-inserts` безопаснее, но восстановление будет ещё медленнее.

`--lock-wait-timeout=`*`время_ожидания`*

Не ждать бесконечно получения разделяемых блокировок таблиц в начале процедуры выгрузки. Вместо этого выдать ошибку, если не удастся заблокировать таблицы за указанное *`время_ожидания`*. Это время можно задать в любом из форматов, принимаемых командой `SET statement_timeout`. Допустимые значения зависят от версии сервера, выгружающего данные, но количество миллисекунд в виде целого числа принимают все версии, начиная с 7.3. Более ранние версии игнорируют этот параметр.

`--no-publications`

Не выгружать публикации.

`--no-role-passwords`

Не выгружать пароли ролей. При восстановлении все роли получат пароль NULL и не смогут пройти проверку подлинности, пока им не будут назначены пароли. Так как значения паролей не нужны, когда используется это указание, информация о ролях считывается из системного представления `pg_roles`, а не из `pg_authid`. Таким образом, данный вариант может быть также полезен, если доступ к `pg_authid` ограничен политикой безопасности.

`--no-security-labels`

Не выгружать метки безопасности.

`--no-subscriptions`

Не выгружать подписки.

`--no-sync`

По умолчанию `pg_dumpall` ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром `pg_dumpall` завершается немедленно, то есть выполняется быстрее, но в случае неожиданного сбоя операционной системы выгруженные данные могут оказаться испорченными. Вообще этот параметр предназначен прежде всего для тестирования, для производственной среды он не подходит.

`--no-tablespaces`

Не выводить команды, создающие или выбирающие табличные пространства для объектов. С этим параметром все объекты будут созданы в пространстве по умолчанию, установленном во время восстановления.

`--no-unlogged-table-data`

Не выгружать содержимое нежурналируемых таблиц. Этот параметр не влияет на то, как выгружаются определения этих таблиц (схема); он отключает только выгрузку данных из них.

`--quote-all-identifiers`

Принудительно экранировать все идентификаторы. Этот параметр рекомендуется при выгрузке базы, когда основная версия сервера PostgreSQL, с которого выгружается база, отличается от версии pg_dumpall, или когда выгруженная копия предназначена для загрузки на сервере с другой основной версией. По умолчанию pg_dumpall экранирует только те идентификаторы, которые являются зарезервированными словами в собственной основной версии. Иногда это приводит к проблемам совместимости с серверами других версий, в которых множество зарезервированных слов может быть несколько другим. Применение параметра `--quote-all-identifiers` предотвращает подобные проблемы, ценой ухудшения читаемости скрипта с выгруженными данными.

`--use-set-session-authorization`

Выводить команды `SET SESSION AUTHORIZATION`, соответствующие стандарту, вместо `ALTER OWNER`, для назначения владельцев объектов. В результате выгруженный скрипт будет более стандартизированным, но может не восстановиться корректно, в зависимости от истории объектов.

`-?`
`--help`

Показать справку по аргументам командной строки pg_dumpall и завершиться.

Далее описаны параметры управления подключением.

`-d` *`строка_подключения`*
`--dbname=`*`строка_подключения`*

Указывает параметры подключения к серверу в формате строки подключения; они будут переопределять любые одноимённые параметры, заданные в командной строке.

Этот параметр называется `--dbname` для согласованности с другими клиентскими приложениями, но так как pg_dumpall подключается не к одной базе данных, имя базы в строке подключения игнорируется. Чтобы указать имя базы данных, через подключение к которой будут выгружаться глобальные объекты и находиться другие выгружаемые базы, воспользуйтесь параметром `-l`.

`-h` *сервер*
`--host=`*сервер*

Указывает имя компьютера, на котором работает сервер баз данных. Если значение начинается с косой черты, оно определяет каталог Unix-сокета. Значение по умолчанию берётся из переменной окружения `PGHOST`, если она установлена. В противном случае выполняется подключение к Unix-сокету.

`-l` *имя_бд*
`--database=`*имя_бд*

Задаёт имя базы данных, через подключение к которой будут выгружаться глобальные объекты и находиться другие выгружаемые базы. По умолчанию используется база `postgres`, а в случае её отсутствия — `template1`.

`-p` *порт*
`--port=`*порт*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной окружения `PGPORT`, если она установлена, либо числом, заданным при компиляции.

`-U` *имя_пользователя*
`--username=`*имя_пользователя*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как pg_dumpall запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, pg_dumpall лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

Заметьте, что пароль будет запрашиваться повторно для выгрузки каждой базы данных. Поэтому обычно лучше настроить файл `~/.pgpass`, и не вводить пароль каждый раз вручную.

`--role=`*имя роли*

Задаёт имя роли, которая будет осуществлять выгрузку. Получив это имя, pg_dumpall выполнит `SET ROLE` *имя_роли* после подключения к базе данных. Это полезно, когда проходящий проверку пользователь (указанный в `-U`) не имеет прав, необходимых для pg_dumpall, но может переключиться на роль, наделённую этими правами. В некоторых окружениях правила запрещают подключаться к серверу непосредственно суперпользователю, и этот параметр позволяет выполнить выгрузку, не нарушая их.

Переменные окружения

`PGHOST`
`PGOPTIONS`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 31.14).

Замечания

Так как pg_dumpall внутри себя вызывает pg_dump, часть диагностических сообщений будет относиться к pg_dump.

После восстановления имеет смысл запустить `ANALYZE` для каждой базы данных, чтобы оптимизатор получил актуальную статистику. Также можно запустить анализ для всех баз данных, выполнив команду `vacuumdb -a -z`.

При использовании pg_dumpall требуется, чтобы все необходимые каталоги табличных пространств существовали до восстановления; в противном случае создание баз данных в нестандартном размещении завершится ошибкой.

Примеры

Выгрузка всех баз данных:

```
$ pg_dumpall > db.out
```

Загрузить базы данных из этого файла можно так:

```
$ psql -f db.out postgres
```

(К какой базе данных вы подключаетесь, здесь не важно, так как скрипт, созданный утилитой pg_dumpall, будет содержать все команды, требующиеся для создания сохранённых баз данных и подключения к ним.)

См. также

Обратитесь к описанию pg_dump, чтобы узнать об условиях, при которых могут возникнуть проблемы.

pg_isready

pg_isready — проверить соединение с сервером Postgres Pro

Синтаксис

`pg_isready` [*параметр-подключения*...] [*параметр*...]

Описание

Утилита pg_isready предназначена для проверки соединения с сервером баз данных Postgres Pro. Результат проверки передаётся в коде завершения.

Параметры

`-d` *имя_бд*
`--dbname=`*имя_бд*

Указывает имя базы данных для подключения. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

`-h` *компьютер*
`--host=`*компьютер*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *порт*
`--port=`*порт*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной среды `PGPORT`, если она установлена, либо числом, заданным при компиляции, обычно 5432.

`-q`
`--quiet`

Не выводить сообщение о состоянии. Это полезно в скриптах.

`-t` *секунды*
`--timeout=`*секунды*

Максимальное время ожидания (в секундах) при попытке подключения, по истечении которого констатируется, что сервер не отвечает. Значение по умолчанию — 3 секунды.

`-U` *имя_пользователя*
`--username=`*имя_пользователя*

Подключиться к базе данных с заданным именем пользователя вместо подразумеваемого по умолчанию.

`-V`
`--version`

Сообщить версию pg_isready и завершиться.

`-?`
`--help`

Показать справку по аргументам командной строки pg_isready и завершиться.

Код завершения

Утилита pg_isready возвращает в оболочку `0`, если сервер принимает подключения, `1`, если он сбрасывает подключения (например, во время загрузки), `2`, если при попытке подключения не получен ответ, и `3`, если попытки подключения не было (например, из-за некорректных параметров).

Переменные окружения

Как и большинство других утилит Postgres Pro, `pg_isready` также использует переменные среды, поддерживаемые libpq (см. Раздел 31.14).

Замечания

Чтобы получить состояние сервера, передавать имя пользователя, пароль и имя базы данных не требуется; но если передать некорректные значения, сервер выведет в журнал сообщение о неудачной попытке подключения.

Примеры

Обычное использование:

```
$ pg_isready
/tmp:5432 - accepting connections
$ echo $?
0
```

Запуск с параметрами подключения, во время загрузки кластера Postgres Pro:

```
$ pg_isready -h localhost -p 5433
localhost:5433 - rejecting connections
$ echo $?
1
```

Запуск с параметрами подключения, в случае, когда кластер Postgres Pro недоступен:

```
$ pg_isready -h someremotehost
someremotehost:5432 - no response
$ echo $?
2
```

pg_receivewal

pg_receivewal — получает журналы предзаписи с сервера Postgres Pro

Синтаксис

`pg_receivewal` [*параметр*...]

Описание

Утилита pg_receivewal предназначена для приёма журнала предзаписи от работающего кластера Postgres Pro. Журнал предзаписи передаётся по протоколу потоковой репликации и записывается в локальный каталог. Затем этот каталог можно использовать в качестве архива для восстановления состояния на момент времени (см. Раздел 24.3).

pg_receivewal принимает журнал предзаписи в реальном времени по мере того, как он генерируется на сервере, и не ждёт завершения сегментов, как это делает archive_command. Поэтому pg_receivewal можно использовать, не устанавливая archive_timeout.

В отличие от приёмника WAL, работающего на ведомом сервере Postgres Pro, pg_receivewal по умолчанию сохраняет на диск данные WAL, только когда файл WAL закрывается. Для сохранения данных WAL в реальном времени необходимо использовать ключ `--synchronous`. Так как приёмник pg_receivewal не применяет WAL, важно не допустить, чтобы он стал синхронным ведомым сервером, когда параметр synchronous_commit равен `remote_apply`. Если это произойдёт, он будет выглядеть как ведомый, который никогда не может нагнать ведущего, что приведёт к блокированию фиксации транзакций. Чтобы это предотвратить, нужно либо установить подходящее значение параметра synchronous_standby_names, либо задать для pg_receivewal такое имя приложения (`application_name`), которое не соответствует установленному имени, либо выбрать для `synchronous_commit` значение, отличное от `remote_apply`.

Журнал предзаписи передаётся через обычное подключение к Postgres Pro, с использованием протокола репликации. Подключение должен устанавливать суперпользователь или пользователь с правом `REPLICATION` (см. Раздел 20.2), а в `pg_hba.conf` должно разрешаться подключение для репликации. Кроме того, параметр max_wal_senders на сервере должен быть достаточно большим, чтобы можно было создать ещё один сеанс для передачи потока.

Начальная точка передачи журнала предзаписи вычисляется при запуске pg_receivewal так:

1. Сначала сканируется каталог, в который помещаются файлы сегментов WAL, в нём выбирается последний завершённый файл сегмента, и начальной точкой считается начало следующего файла сегмента WAL.
2. Если вычислить начальную точку предыдущим способом не удаётся, используется последняя позиция сохранённых данных в WAL, которую выдаёт команда `IDENTIFY_SYSTEM`.

Если подключение разорвалось или его с самого начала не удаётся установить из-за некритической ошибки, pg_receivewal будет бесконечно повторять попытки подключения и восстановит передачу, как только сможет. Чтобы отменить это поведение, воспользуйтесь параметром `-n`.

Параметры

`-D` *каталог*
`--directory=`*каталог*

Каталог, в который будут записываться данные.

Этот параметр является обязательным.

`--if-not-exists`

Не выдавать ошибку, когда указан параметр `--create-slot` и слот с заданным именем уже существует.

`-n`
`--no-loop`

Не повторять цикл при ошибках подключения, а сразу завершать работу, возвращая ошибку.

`-s` *интервал*
`--status-interval=`*интервал*

Указывает интервал в секундах между отправками серверу пакетов состояния. Это позволяет упростить мониторинг прогресса. Чтобы выключить периодическое обновление состояния, необходимо установить значение в ноль. При этом обновление будет отправляться по запросу сервера для избежания отсоединения по истечению времени. Значение по умолчанию составляет 10 секунд.

`-S` *имя_слота*
`--slot=`*имя_слота*

Указывает pg_receivewal использовать существующий слот репликации (см. Подраздел 25.2.6). Когда задан этот параметр, pg_receivewal будет сообщать серверу текущую позицию сохранения, отмечая, какой сегмент был сохранён на диске, чтобы сервер мог удалить этот сегмент, если он больше не нужен.

Когда клиент репликации pg_receivewal настроен на сервере как синхронный ведомый сервер, для используемого слота репликации серверу будет передаваться позиция сохранённых данных, но только когда файл WAL закрывается. Таким образом, в такой конфигурации транзакции на ведущем сервере будут ожидать завершения продолжительное время и по сути будут работать неудовлетворительно. Чтобы эта конфигурация работала корректно, нужно дополнительно указать параметр `--synchronous` (см. ниже).

`--synchronous`

Сохранять данные WAL на диск сразу после того, как они были получены. Также передавать пакет состояния сразу после сохранения, вне зависимости от `--status-interval`.

Этот параметр следует указывать, если клиент репликации pg_receivewal настроен на сервере как синхронный ведомый, чтобы обеспечить своевременную передачу ответа серверу.

`-v`
`--verbose`

Включает режим подробных сообщений.

`-Z` *уровень*
`--compress=`*уровень*

Включает gzip-сжатие журналов предзаписи и задаёт уровень сжатия от 0 (без сжатия) до 9 (максимальное сжатие). При этом ко всем именам файлов tar добавляется суффикс `.gz`.

Далее описаны параметры управления подключением.

`-d` *строка_подключения*
`--dbname=`*строка_подключения*

Указывает параметры подключения к серверу в формате строки подключения; они будут переопределять любые одноимённые параметры, заданные в командной строке.

Параметр называется `--dbname` для согласованности с другими клиентскими приложениями, но так как pg_receivewal не подключается к какой-либо конкретной базе, это имя в строке подключения игнорируется.

`-h` *сервер*
`--host=`*сервер*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета. Значение по умолчанию берётся из переменной окружения `PGHOST`, если она установлена. В противном случае выполняется подключение к Unix-сокету.

`-p` *порт*
`--port=`*порт*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной окружения `PGPORT`, если она установлена, либо числом, заданным при компиляции.

`-U` *имя_пользователя*
`--username=`*имя_пользователя*

Имя пользователя для подключения.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как pg_receivewal запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, pg_receivewal лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

pg_receivewal может выполнить одно из двух действий в отношении слотов физической репликации:

`--create-slot`

Создать слот физической репликации с именем, заданным аргументом `--slot`, и завершиться.

`--drop-slot`

Удалить слот репликации с именем, заданным аргументом `--slot`, и завершиться.

Другие флаги:

`-V`
`--version`

Сообщить версию pg_receivewal и завершиться.

`-?`
`--help`

Вывести справку по аргументам командной строки pg_receivewal и завершиться.

Переменные окружения

Эта утилита, как и большинство других утилит Postgres Pro, использует переменные среды, поддерживаемые libpq (см. Раздел 31.14).

Замечания

Применяя pg_receivewal вместо archive_command в качестве основного способа резервного копирования WAL, настоятельно рекомендуется использовать слоты репликации. В противном случае сервер вполне может переписать или удалить файлы журнала предзаписи, прежде чем они будут скопированы, так как он не получает никакой информации, через archive_command или слоты репликации, о том, как проходит архивация потока WAL. Учтите, однако, что при использовании слота репликации может заполниться всё место на диске, если принимающая сторона не будет успевать принимать данные WAL.

Примеры

Следующая команда принимает журнал предзаписи с сервера `mydbserver` и сохраняет его в локальном каталоге `/usr/local/pgsql/archive`:

```
$ pg_receivewal -h mydbserver -D /usr/local/pgsql/archive
```

См. также

pg_basebackup

pg_recvlogical

pg_recvlogical — управление потоками логического декодирования Postgres Pro

Синтаксис

`pg_recvlogical` [*параметр*...]

Описание

Утилита `pg_recvlogical` управляет слотами логического декодирования и принимает данные из таких слотов репликации.

Она создаёт соединение в режиме репликации, так что на него распространяются те же ограничения, что и с pg_receivewal, плюс ограничения логической репликации (см. Главу 46).

`pg_recvlogical` не предоставляет возможностей, соответствующих режимам peek и get в SQL-интерфейсе логического декодирования. Она передаёт серверу подтверждения воспроизведения данных по мере их получения и при штатном выходе. Чтобы просмотреть данные, ожидающие передачи через слот, не принимая их, воспользуйтесь функцией `pg_logical_slot_peek_changes`.

Параметры

Для выбора действия необходимо указать минимум один из этих параметров:

`--create-slot`

Создать новый слот логической репликации с именем, заданным аргументом `--slot`, используя модуль вывода, заданный аргументом `--plugin`, для базы данных, указанной в `--dbname`.

`--drop-slot`

Удалить слот репликации с именем, заданным аргументом `--slot`, и завершиться.

`--start`

Начать приём потока изменений из слота логической репликации с именем, заданным аргументом `--slot`, и продолжать до сигнала прерывания. Если передача потока прерывается на другой стороне из-за выключения или остановки сервера, цикл подключения и передачи повторяется (если не добавлен параметр `--no-loop`).

Формат потока определяется модулем вывода, выбранным при создании слота.

Для получения потока подключаться нужно к той же базе, для которой создавался слот.

Параметры `--create-slot` и `--start` исключают друг друга. Действие `--drop-slot` несовместимо с любыми другими действиями.

Следующие параметры командной строки управляют расположением и форматом выводимых данных, а также другим поведением репликации:

`-E` *lsn*
`--endpos=`*lsn*

В режиме `--start` автоматически закончить репликацию и выйти с кодом обычного завершения 0, когда при приёме данных достигается указанный LSN. Если этот ключ указывается не в режиме `--start`, выдаётся ошибка.

Если встречается запись с LSN, в точности равным *lsn*, эта запись будет выведена.

С указанием `--endpos` границы транзакций не отслеживаются, так что вывод программы может оказаться обрезанным посередине транзакции. Частично полученная транзакция не будет

считаться принятой и будет воспроизведена заново при следующем чтении из этого слота. Отдельные сообщения не обрезаются никогда.

`-f` *имя_файла*
`--file=`*имя_файла*

Записывать полученные и декодированные данные транзакций в указанный файл. Для вывода в stdout укажите – (минус).

`-F` *секунды*
`--fsync-interval=`*секунды*

Устанавливает, как часто pg_recvlogical будет вызывать `fsync()`, чтобы гарантировать, что выходной файл надёжно сохранён на диске.

Сервер время от времени даёт клиенту команду сохранить данные и сообщить сохранённую позицию, но этот параметр позволяет выполнять сохранение чаще.

При значении, равном `0`, функция `fsync()` вообще не вызывается, но серверу сообщается новая позиция. Это может привести к потере данных в случае сбоя.

`-I` *lsn*
`--startpos=`*lsn*

В режиме `--start` репликация начнётся с данного LSN. Как это работает, подробно описывается в Главе 46 и Разделе 50.4. В других режимах игнорируется.

`--if-not-exists`

Не выдавать ошибку, когда указан параметр `--create-slot` и слот с заданным именем уже существует.

`-n`
`--no-loop`

Когда подключение к серверу потеряно, не повторять цикл, просто завершить работу.

`-o` *имя*`[=`*значение*`]`
`--option=`*имя*`[=`*значение*`]`

Передаёт параметр *имя_параметра* модулю вывода, при этом может быть передано и его *значение*. Набор параметров и их действия зависят от выбранного модуля вывода.

`-P` *модуль*
`--plugin=`*модуль*

Использовать указанный модуль вывода логического декодирования при создании слота. См. Главу 46. Этот параметр не действует, если слот уже существует.

`-s` *секунды*
`--status-interval=`*секунды*

Этот параметр действует так же, как одноимённый параметр pg_receivewal (см. его описание там).

`-S` *имя_слота*
`--slot=`*имя_слота*

Этот параметр задаёт имя слота логической репликации, который будет использоваться в режиме `--start`, создаваться в режиме `--create-slot` или удаляться в режиме `--drop-slot`.

`-v`
`--verbose`

Включает режим подробных сообщений.

Далее описаны параметры управления подключением.

`-d` *имя_бд*
`--dbname=`*имя_бд*

Имя базы данных для подключения. Как именно используется данная база, рассказывается в описании действий программы. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке. По умолчанию в качестве имени базы выбирается имя пользователя.

`-h` *имя_компьютера-или-ip*
`--host=`*имя_компьютера-или-ip*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета. Значение по умолчанию берётся из переменной окружения `PGHOST`, если она установлена. В противном случае выполняется подключение к Unix-сокету.

`-p` *порт*
`--port=`*порт*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной окружения `PGPORT`, если она установлена, либо числом, заданным при компиляции.

`-U` *user*
`--username=`*user*

Имя пользователя для подключения. По умолчанию это имя текущего пользователя операционной системы.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как pg_recvlogical запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, pg_recvlogical лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

Также есть следующие дополнительные параметры:

`-V`
`--version`

Сообщить версию pg_recvlogical и завершиться.

`-?`
`--help`

Показать справку по аргументам командной строки pg_recvlogical и завершиться.

Переменные окружения

Как и большинство других утилит Postgres Pro, приложение также использует переменные окружения, поддерживаемые libpq (см. Раздел 31.14).

Примеры

Примеры использования можно найти в Разделе 46.1.

См. также

pg_receivewal

pg_restore

pg_restore — восстановить базу данных Postgres Pro из файла архива, созданного командой pg_dump

Синтаксис

`pg_restore` [*параметр-подключения*...] [*параметр*...] [*имя_файла*]

Описание

Утилита pg_restore предназначена для восстановления базы данных Postgres Pro из архива, созданного командой pg_dump в любом из не текстовых форматов. Она выполняет команды, необходимые для восстановления того состояния базы данных, в котором база была сохранена. При наличии файлов архивов, pg_restore может восстанавливать данные избирательно или даже переупорядочить объекты перед восстановлением. Заметьте, что разработанный для файлов архива формат не привязан к архитектуре.

Утилита pg_restore может работать в двух режимах. Если указывается имя базы данных, pg_restore подключается к этой базе данных и восстанавливает содержимое архива непосредственно в неё. В противном случае создаётся SQL-скрипт с командами, необходимыми для пересоздания базы данных, который затем выводится в файл или в стандартное устройство вывода. Сформированный скрипт будет в точности соответствовать выводу pg_dump в простом текстовом формате. Поэтому некоторые из параметров, управляющих выводом, аналогичны параметрам pg_dump.

Разумеется, pg_restore может восстановить информацию, только если она присутствует в файле архива, и только в существующем виде. Например, если архив был создан с указанием «выгрузить данные в виде команд `INSERT`», pg_restore не сможет загрузить эти данные, используя операторы `COPY`.

Параметры

Утилита pg_restore принимает следующие аргументы командной строки.

имя_файла

Указывает расположение восстанавливаемого файла архива (или каталога, для архива в формате каталога). По умолчанию используется устройство стандартного ввода.

`-a`
`--data-only`

Восстанавливать только данные, без схемы (определений данных). При этом восстанавливаются данные таблиц, большие объекты и значения последовательностей, имеющиеся в архиве.

Флаг похож на `--section=data`, но по историческим причинам не равнозначен ему.

`-c`
`--clean`

Удалить (DROP) объекты базы данных, прежде чем пересоздавать их. (Без дополнительного указания `--if-exists` при этом могут выдаваться безвредные сообщения об ошибках, если таких объектов не окажется в целевой базе данных.)

`-C`
`--create`

Создать базу данных, прежде чем восстанавливать данные. Если также указан параметр `--clean`, удалить и пересоздать целевую базу данных перед подключением к ней.

С этим параметром база, заданная параметром `-d`, применяется только для подключения и выполнения начальных команд `DROP DATABASE` и `CREATE DATABASE`. Все данные восстанавливаются в базу данных, имя которой записано в архиве.

`-d` *имя_бд*
`--dbname=`*имя_бд*

Подключиться к базе данных *имя_базы* и восстановить данные непосредственно в неё. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

`-e`
`--exit-on-error`

Завершать работу в случае возникновения ошибки при выполнении команд SQL в базе данных. По умолчанию процесс восстановления продолжается, а по его окончании выдаётся число ошибок.

`-f` *имя_файла*
`--file=`*имя_файла*

Задаёт файл для вывода сгенерированного скрипта или списка объектов, получаемого с параметром `-l`. Чтобы выбрать стандартное устройство вывода, используйте `-` (этот вариант также подразумевается по умолчанию).

`-F` *формат*
`--format=`*формат*

Задаёт формат архива. Указывать формат необязательно, так как pg_restore определяет формат автоматически. Но если формат задаётся, допускается один из этих вариантов:

`c`
`custom`

Архив сохранён в специальном формате pg_dump.

`d`
`directory`

Архив сохранён в каталоге.

`t`
`tar`

Архив сохранён в формате `tar`.

`-I` *индекс*
`--index=`*индекс*

Восстановить определение только заданного индекса. Добавив дополнительные ключи `-I`, можно указать несколько индексов.

`-j` *число-заданий*
`--jobs=`*число-заданий*

Запустить наиболее длительные операции pg_restore (в частности, загрузку данных, создание индексов или ограничений) в нескольких параллельных заданиях. Это позволяет кардинально сократить время восстановления большой базы данных, когда сервер работает на многопроцессорном компьютере.

Каждое задание выполняется в отдельном задании или потоке, в зависимости от операционной системы, и использует отдельное подключение к серверу.

Оптимальное значение этого параметра зависит от аппаратной конфигурации сервера, клиента и сети. В частности, имеет значение количество процессорных ядер и устройство дискового

хранилища. В качестве начального значения можно выбрать число ядер на сервере, но и при увеличении этого значения во многих случаях восстановление будет быстрее. Конечно, при слишком больших значениях производительность упадёт из-за перегрузки.

Этот параметр поддерживается только с архивом в специальном формате или в каталоге. Входные данные должны поступать из обычного файла или каталога (а не, например из канала ввода). Данный параметр игнорируется, когда генерируется скрипт (нет прямого подключения к базе данных). Кроме того, несколько заданий не могут выполняться в сочетании с параметром `--single-transaction`.

`-l`
`--list`

Вывести оглавление архива. Вывод этой операции можно подать на вход этой же команде с параметром `-L`. Учтите, что когда вместе с `-l` применяются параметры фильтрации (например, `-n` или `-t`), они сокращают список выводимых объектов.

`-L` *файл-список*
`--use-list=`*файл-список*

Восстановить из архива только элементы, перечисленные в *файле-списке*, и в том порядке, в каком они идут в этом файле. Заметьте, что когда вместе с `-L` применяются параметры фильтрации (например, `-n` или `-t`), они дополнительно ограничивают восстанавливаемые объекты.

Данный *файл-список* обычно представляет собой отредактированный результат предыдущей операции `-l`. Строки в нём могут быть переставлены или удалены, а также могут быть закомментированы точкой с запятой (`;`), добавленной в начале строки. См. примеры ниже.

`-n` *схема*
`--schema=`*схема*

Восстановить только объекты в указанной схеме. Добавив дополнительные ключи `-n`, можно указать несколько схем. Этот параметр можно сочетать с `-t`, чтобы восстановить только определённую таблицу.

`-N` *схема*
`--exclude-schema=`*схема*

Не восстанавливать объекты в указанной схеме. Добавив дополнительные ключи `-N`, можно исключить несколько схем.

Когда и с ключом `-n`, и с ключом `-N` передаётся имя одной схемы, ключ `-N` выигрывает и схема исключается.

`-O`
`--no-owner`

Не генерировать команды, устанавливающие владение объектами, как в исходной базе данных. По умолчанию, pg_restore генерирует команды `ALTER OWNER` или `SET SESSION AUTHORIZATION`, восстанавливающие исходных владельцев создаваемых элементов схемы. Однако эти команды можно будет выполнить, только если к базе данных первоначально подключается суперпользователь (или пользователь, владеющими всеми объектами в скрипте). Чтобы получить скрипт, который сможет восстановить любой подключающийся пользователь (но при этом он станет владельцем всех созданных объектов), используется `-O`.

`-P` *имя-функции(тип-аргумента[, ...])*
`--function=`*имя-функции(тип-аргумента[, ...])*

Восстановить только указанную функцию. При этом важно записать имя функции и аргументы в точности так, как они фигурируют в оглавлении файла архива. Добавив дополнительные ключи `-P`, можно указать несколько функций.

`-R`
`--no-reconnect`

Параметр является устаревшим, но в целях совместимости ещё работает.

`-s`
`--schema-only`

Восстановить только схему (определения данных), без данных, в объёме, в котором элементы схемы представлены в архиве.

Действие параметра противоположно действию `--data-only`. Это похоже на указание `--section=pre-data --section=post-data`, но по историческим причинам не равнозначно ему.

(Не путайте этот параметр с `--schema`, где слово «схема» используется в другом значении.)

`-S` *имя_пользователя*
`--superuser=`*имя_пользователя*

Задаёт имя суперпользователя, полномочия которого будут использоваться для отключения триггеров. Этот параметр применяется только с параметром `--disable-triggers`.

`-t` *таблица*
`--table=`*таблица*

Восстановить определение и/или данные только указанной таблицы. В этом контексте под «таблицей» подразумеваются также представления, материализованные представления, последовательности и сторонние таблицы. Чтобы выбрать несколько таблиц, ключ `-t` можно указать несколько раз. Этот параметр можно скомбинировать с `-n`, чтобы выбрать таблицу(ы) в определённой схеме.

> **Примечание**
>
> Когда указывается `-t`, pg_restore не пытается восстанавливать объекты базы данных, от которых могут зависеть выбранные таблицы. Таким образом, в этом случае не гарантируется, что выгруженные таблицы будут успешно восстановлены в чистой базе данных.

> **Примечание**
>
> Этот флаг действует не совсем так, как флаг `-t` утилиты pg_dump. В настоящее время pg_restore не поддерживает выбор объектов по маске, а также не позволяет указать имя схемы с `-t`.

> **Примечание**
>
> В версиях Postgres Pro до 9.6 этот флаг выбирал только таблицы, но не другие типы отношений.

`-T` *триггер*
`--trigger=`*триггер*

Восстановить только указанный триггер. Добавив дополнительные ключи `-T`, можно указать несколько триггеров.

`-v`
`--verbose`

Включает режим подробных сообщений.

`-V`
`--version`

Сообщить версию pg_restore и завершиться.

`-x`
`--no-privileges`
`--no-acl`

Не восстанавливать права доступа (не выполнять команды GRANT/REVOKE).

`-1`
`--single-transaction`

Произвести восстановление в одной транзакции (то есть, завернуть выполняемые команды в `BEGIN`/`COMMIT`). При этом гарантируется, что либо все команды будут выполнены успешно, либо не будет никаких изменений. Этот режим подразумевает `--exit-on-error`.

`--disable-triggers`

Этот параметр действует только при выгрузке одних данных. С ним pg_restore выполняет команды, отключающие триггеры в целевых таблицах на время загрузки данных. Используйте его, если в ваших таблицах определены проверки ссылочной целостности или другие триггеры, которые вы не хотели бы выполнять в процессе загрузки данных.

В настоящее время команды, генерируемые с `--disable-triggers`, должны выполнятся суперпользователем. Поэтому необходимо также задать имя суперпользователя в параметре `-S` или, что предпочтительнее, запускать pg_restore от имени суперпользователя Postgres Pro.

`--enable-row-security`

Этот параметр имеет смысл только при восстановлении содержимого таблицы, для которой включена защита строк. По умолчанию pg_restore устанавливает для row_security значение off для уверенности, что в таблице восстановлены все данные. Если у пользователя недостаточно прав для обхода защиты строк, выдаётся ошибка. Этот параметр указывает pg_restore установить в row_security значение on, чтобы пользователь мог попытаться восстановить содержимое таблицы с включённой защитой строк. Однако и при этом возможна ошибка, если пользователь не будет иметь права добавлять в эту таблицу выгруженные строки данных.

Заметьте, что в настоящее время для этого требуется, чтобы выгрузка выполнялась в режиме `INSERT`, так как `COPY FROM` не поддерживает защиту строк.

`--if-exists`

При очистке целевой базы использовать условные команды (добавлять предложение `IF EXISTS`). Применяется только с параметром `--clean`.

`--no-data-for-failed-tables`

По умолчанию данные восстанавливаются даже при ошибке команды создания таблицы (например, когда она уже существует). С этим параметром данные в таком случае не восстанавливаются. Это поведение полезно, если в целевой таблице уже содержатся нужные данные. Например, вспомогательные таблицы для расширений Postgres Pro (в частности, PostGIS) могут быть уже заполнены; этот параметр позволяет предотвратить дублирование или загрузку устаревших данных в эти таблицы.

Этот параметр действует только при восстановлении непосредственно в базу данных (не при генерации SQL-скрипта).

--no-publications

Не выводить команды, восстанавливающие публикации, даже если они содержатся в архиве.

--no-security-labels

Не выводить команды, восстанавливающие метки безопасности, даже если они содержатся в архиве.

--no-subscriptions

Не выводить команды, восстанавливающие подписки, даже если они содержатся в архиве.

--no-tablespaces

Не формировать команды для указания табличных пространств. При восстановлении все объекты будут создаваться в табличном пространстве по умолчанию.

--section=*имя секции*

Восстановить только указанный раздел. В качестве имени раздела можно задать pre-data, data или post-data. Указав этот параметр неоднократно, можно выбрать несколько разделов. По умолчанию восстанавливаются все разделы.

Раздел «data» содержит собственно данные таблиц и определения больших объектов. В разделе «post-data» содержатся определения индексов, триггеров, правил и ограничений (кроме отдельно проверяемых). Раздел «pre-data» содержит все остальные определения.

--strict-names

Требует, чтобы каждому указанию схемы (-n/--schema) и таблицы (-t/--table) соответствовала минимум одна схема/таблица в файле резервной копии.

--use-set-session-authorization

Выводить команды SET SESSION AUTHORIZATION, соответствующие стандарту, вместо ALTER OWNER, для назначения владельцев объектов. В результате выгруженный скрипт будет более стандартизированным, но может не восстановиться корректно, в зависимости от истории объектов.

-?
--help

Показать справку по аргументам командной строки pg_restore и завершиться.

pg_restore также принимает в качестве параметров соединения следующие аргументы командной строки:

-h *сервер*
--host=*сервер*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета. Значение по умолчанию берётся из переменной окружения PGHOST, если она установлена. В противном случае выполняется подключение к Unix-сокету.

-p *порт*
--port=*порт*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной окружения PGPORT, если она установлена, либо числом, заданным при компиляции.

-U *имя_пользователя*
--username=*имя_пользователя*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как pg_restore запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, pg_restore лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

`--role=`*`имя роли`*

Задаёт имя роли, которая будет осуществлять восстановление. Получив это имя, pg_restore выполнит `SET ROLE` *`имя_роли`* после подключения к базе данных. Это полезно, когда проходящий проверку пользователь (указанный в `-U`) не имеет прав, необходимых для pg_restore, но может переключиться на роль, наделённую этими правами. В некоторых окружениях правила запрещают подключаться к серверу непосредственно суперпользователю, и этот параметр позволяет выполнить восстановление, не нарушая их.

Переменные окружения

`PGHOST`
`PGOPTIONS`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию

Как и большинство других утилит Postgres Pro, эта утилита также использует переменные среды, поддерживаемые libpq (см. Раздел 31.14). Однако она не учитывает `PGDATABASE`, когда имя базы не указано.

Диагностика

Когда с параметром `-d` устанавливается прямое подключение к базе данных, pg_restore выполняет обычные операторы SQL. При этом применяются все свойства подключения по умолчанию и переменные окружения, которые использует клиентская библиотека libpq. Если вы сталкиваетесь с проблемами при запуске pg_restore, убедитесь в том, что вы можете получить информацию из базы данных, используя, например psql.

Замечания

Если в вашей инсталляции база данных `template1` содержит какие-либо дополнения, важно убедиться в том, что вывод pg_restore загружается в действительно пустую базу; иначе вы, скорее всего, получите ошибки из-за дублирования определений создаваемых объектов. Чтобы получить пустую базу данных без дополнительных объектов, выберите в качестве шаблона `template0`, а не `template1`, например так:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

Ограничения pg_restore описаны ниже.

- При восстановлении данных в уже существующие таблицы с параметром `--disable-triggers`, pg_restore выполняет команды, отключающие триггеры в пользовательских таблицах

до добавления данных, а затем, после добавления данных, выполняет команды, снова включающие эти триггеры. Если восстановление прервётся в середине, системные каталоги могут оказаться в некорректном состоянии.

- Утилита pg_restore не способна восстанавливать большие объекты избирательно; например, только для определённой таблицы. Если архив содержит большие объекты, они будут восстановлены все, либо не будут восстановлены никакие (если они были исключены параметрами `-L`, `-t` и т. п.).

Также обратитесь к документации pg_dump, чтобы узнать о связанных ограничениях pg_dump.

После восстановления имеет смысл запустить `ANALYZE` для каждой восстановленной таблицы, чтобы оптимизатор получил актуальную статистику; за дополнительными сведениями обратитесь к Подразделу 23.1.3 и Подразделу 23.1.6.

Примеры

Предположим, что мы выгрузили базу данных `mydb` в файл специального формата:

```
$ pg_dump -Fc mydb > db.dump
```

Мы можем удалить базу данных и восстановить её из копии:

```
$ dropdb mydb
$ pg_restore -C -d postgres db.dump
```

В аргументе `-d` можно указать любую базу данных, существующую в кластере; pg_restore использует её, только чтобы выполнить команду `CREATE DATABASE` для базы `mydb`. С параметром `-C` данные всегда восстанавливаются в базу, имя которой записано в файле архива.

Восстановить данные в новую базу `newdb` можно так:

```
$ createdb -T template0 newdb
$ pg_restore -d newdb db.dump
```

Обратите внимание, мы не используем параметр `-C`, а вместо этого подключаемся непосредственно к базе, в которую хотим восстановить данные. Также заметьте, что мы создаём базу данных из шаблона `template0`, а не `template1`, чтобы изначально она была гарантированно пустой.

Чтобы переупорядочить элементы базы данных, сначала необходимо получить оглавление архива:

```
$ pg_restore -l db.dump > db.list
```

Файл оглавления содержит заголовок и по одной строке для каждого элемента, например:

```
;
; Archive created at Mon Sep 14 13:55:39 2009
;     dbname: DBDEMOS
;     TOC Entries: 81
;     Compression: 9
;     Dump Version: 1.10-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 8.3.5
;     Dumped by pg_dump version: 8.3.8
;
;
; Selected TOC Entries:
;
3; 2615 2200 SCHEMA - public pasha
1861; 0 0 COMMENT - SCHEMA public pasha
```

```
1862; 0 0 ACL - public pasha
317; 1247 17715 TYPE public composite pasha
319; 1247 25899 DOMAIN public domain0 pasha
```

С точки с запятой начинаются комментарии, а число в начале строки обозначает внутренний идентификатор, назначаемый каждому элементу в архиве.

Строки в этом файле можно закомментировать, удалить или переместить. Например, список:

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

можно передать утилите pg_restore, чтобы восстановить только элементы 10 и 6 в указанном порядке:

```
$ pg_restore -L db.list db.dump
```

См. также

pg_dump, pg_dumpall, psql

pg-wrapper

pg-wrapper — управление символическими ссылками Postgres Pro

Синтаксис

`pg-wrapper links { update | remove }`

Описание

Программа pg-wrapper представляет собой скрипт оболочки, включённый в дистрибутив Postgres Pro и предназначенный для управления символическими ссылками на программы Postgres Pro и подключения страниц man по SQL в системах Linux. Так как Postgres Pro устанавливается в `/opt/pgpro/std-10`, это необходимо, чтобы клиентские и серверные программы находились в стандартном системном пути (`PATH`), а также находились новые страницы man по SQL. Эти действия не производятся автоматически во время установки (исключение составляет пакет быстрой установки `postgrespro-std-10`) во избежание возможных конфликтов с другими продуктами на базе PostgreSQL, которые могут уже быть установлены.

Скрипт pg-wrapper входит в состав пакета `postgrespro-std-10-client`. Установив этот пакет, вы сможете найти pg-wrapper в каталоге *`каталог-инсталляции`*`/bin`, где *`каталог-инсталляции`* обозначает путь `/opt/pgpro/std-10`.

Скрипт pg-wrapper должен запускать пользователь root.

Параметры

pg-wrapper принимает следующие аргументы командной строки:

`links` {`update` | `remove`}

Управление символическими ссылками Postgres Pro и страницами man по SQL:

- `update` — создать или заменить символические ссылки для серверных или клиентских программ, распространяемых в составе Postgres Pro, а также подключить страницы man по SQL в файле конфигурации страниц man.
- `remove` — удалить символические ссылки и поддержку страниц man по SQL для текущей версии Postgres Pro.

Замечания

Если вы устанавливаете Postgres Pro из пакета `postgrespro-std-10`, pg-wrapper вызывается автоматически.

Если вы устанавливаете отдельные пакеты Postgres Pro, вы можете запустить этот скрипт вручную, чтобы создать символические ссылки на предоставляемые клиентские и серверные программы, а также подключить страницы man по SQL в файле конфигурации страниц man.

При параллельной установке с другими продуктами на базе `PostgreSQL` поведение `pg-wrapper` зависит от того, поддерживается ли механизм update-alternatives вашей системой и уже установленным продуктом на базе `PostgreSQL`:

- Если поддерживается механизм update-alternatives, `pg-wrapper` добавляет символические ссылки на программы Postgres Pro в систему альтернатив, в соответствии с их приоритетом, а также подключает страницы man по SQL в файл конфигурации страницы man. Информацию об изменении приоритетов в системе альтернатив вы найдёте в руководстве man update-alternatives в вашей системе.

  Если в системе уже установлены страницы man по SQL от другого продукта, от новой инсталляции будут выводиться только новые страницы, а все совпадающие страницы man

— от старой. Чтобы новая документация SQL выводилась для всех страниц, вам потребуется изменить системную конфигурацию, например, поменять значение `MANPATH`.

- Если механизм update-alternatives не поддерживается, `pg_wrapper` изменяет системную конфигурацию, только если отсутствуют конфликты с любыми другими установленными программами или страницами man. В случае таких конфликтов pg-wrapper не создаёт и не изменяет ссылки программ и пропускает интеграцию страниц man по SQL. В этом случае вы можете либо продолжать использовать ранее установленные версии программ и страниц man по SQL, либо изменить параметры `PATH` и `MANPATH` вручную.

Установка из двоичных пакетов в Linux подробно описана в Разделе 16.1.

psql

psql — Интерактивный терминал Postgres Pro

Синтаксис

`psql` [*параметр*...] [*имя_бд* [*имя_пользователя*]]

Описание

Программа psql — это терминальный клиент для работы с Postgres Pro. Она позволяет интерактивно вводить запросы, передавать их в Postgres Pro и видеть результаты. Также запросы могут быть получены из файла или из аргументов командной строки. Кроме того, psql предоставляет ряд метакоманд и различные возможности, подобные тем, что имеются у командных оболочек, для облегчения написания скриптов и автоматизации широкого спектра задач.

Параметры

`-a`
`--echo-all`

Отправляет в стандартный вывод все непустые входные строки по мере их чтения. (Это не относится к строкам, считанным в интерактивном режиме.) Эквивалентно установке переменной `ECHO` в значение `all`.

`-A`
`--no-align`

Переключение на невыровненный режим вывода. (По умолчанию, наоборот, используется выровненный режим вывода.) Равнозначно команде `\pset format unaligned`.

`-b`
`--echo-errors`

Выводит все команды SQL с ошибками в стандартный канал ошибок. Равнозначно присваиванию переменной `ECHO` значения `errors`.

`-c` *команда*
`--command=`*команда*

Передаёт psql *команду* для выполнения. Этот ключ можно повторять и комбинировать в любом порядке с ключом `-f`. Когда указывается `-c` или `-f`, psql не читает команды со стандартного ввода; вместо этого она завершается сразу после обработки всех ключей `-c` и `-f` по порядку.

Заданная *команда* должна быть либо командной строкой, которая полностью интерпретируется сервером (т. е. не использует специфические функции psql), либо одиночной командой с обратной косой чертой. Таким образом, используя `-c`, нельзя смешивать метакоманды SQL и psql. Но это можно сделать, передав несколько ключей `-c` или передав строку в psql через канал:

```
psql -c '\x' -c 'SELECT * FROM foo;'
```

или

```
echo '\x \\ SELECT * FROM foo;' | psql
```

(`\\` — разделитель метакоманд.)

Каждая строка SQL-команд, заданная ключом `-c`, передаётся на сервер как один запрос. Поэтому сервер выполняет её в одной транзакции, даже когда эта строка содержит несколько команд SQL, если только в ней не содержатся явные команды `BEGIN`/`COMMIT`, разделяющие её

на несколько транзакций. Кроме того, psql печатает результат только последней SQL-команды в строке. Это отличается от поведения, когда та же строка считывается из файла или подаётся на стандартный ввод psql, так как в последнем случае psql передаёт каждую команду SQL отдельно.

Из-за такого поведения указание нескольких команд в одной строке `-c` часто приводит к неожиданным результатам. Поэтому лучше использовать несколько ключей `-c` или подавать команды на стандартный ввод psql, применяя либо echo, как показано выше, либо создаваемый прямо в оболочке документ, например:

```
psql <<EOF
\x
SELECT * FROM foo;
EOF
```

`-d` *имя_бд*
`--dbname=`*имя_бд*

Указывает имя базы данных для подключения. Равнозначно указанию *имя_бд* в первом аргументе, не являющемся ключом, в командной строке. Вместо имени может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

`-e`
`--echo-queries`

Посылает все команды SQL, отправленные на сервер, ещё и на стандартный вывод. Эквивалентно установке переменной `ECHO` в значение `queries`.

`-E`
`--echo-hidden`

Отображает фактические запросы, генерируемые `\d` и другими командами, начинающимися с `\`. Это можно использовать для изучения внутренних операций в psql. Эквивалентно установке переменной `ECHO_HIDDEN` значения `on`.

`-f` *имя_файла*
`--file=`*имя_файла*

Читает команды из файла *имя_файла*, а не из стандартного ввода. Этот ключ можно повторять и комбинировать в любом порядке с ключом `-c`. Если указан ключ `-c` или `-f`, программа psql не читает команды со стандартного ввода; вместо этого она завершается после обработки всех ключей `-c` и `-f` по очереди. Не считая этого, данный ключ по большому счёту равнозначен метакоманде `\i`.

Если *имя_файла* задано символом `-` (минус), считывается стандартный ввод до признака конца файла или до метакоманды `\q`. Это позволяет перемежать интерактивный ввод с вводом из файлов. Однако заметьте, что Readline в этом случае не применяется (так же, как и с ключом `-n`).

Использование этого параметра немного отличается от `psql < `*имя_файла*. В основном, оба варианта будут делать то, что вы ожидаете, но с `-f` доступны некоторые полезные свойства, такие как сообщения об ошибках с номерами строк. Также есть небольшая вероятность, что запуск в таком режиме будет быстрее. С другой стороны, вариант с перенаправлением ввода из командного интерпретатора (в теории) гарантирует получение точно такого же вывода, какой вы получили бы, если бы ввели всё вручную.

`-F` *разделитель*
`--field-separator=`*разделитель*

Использование *разделитель* в качестве разделителя полей при невыровненном режиме вывода. Эквивалентно `\pset fieldsep` или `\f`.

`-h` *компьютер*
`--host=`*компьютер*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-H`
`--html`

Включает табличный вывод в формате HTML. Эквивалентно `\pset format html` или команде `\H`.

`-l`
`--list`

Выводит список всех доступных баз данных и завершает работу. Другие параметры, не связанные с соединением, игнорируются. Это похоже на метакоманду `\list`.

Когда используется этот аргумент, psql будет подключаться к базе данных `postgres`, если только в командной строке не задана другая база данных (в параметре `-d` или не через параметры, а, например, через запись службы, но не через переменную окружения).

`-L` *имя_файла*
`--log-file=`*имя_файла*

В дополнение к обычному выводу, записывает вывод результатов всех запросов в файл *имя_файла*.

`-n`
`--no-readline`

Отключает использование Readline для редактирования командной строки и использования истории команд. Может быть полезно для выключения расширенных действий клавиши табуляции при вырезании и вставке.

`-o` *имя_файла*
`--output=`*имя_файла*

Записывает вывод результатов всех запросов в файл *имя_файла*. Эквивалентно команде `\o`.

`-p` *порт*
`--port=`*порт*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной среды `PGPORT`, если она установлена, либо числом, заданным при компиляции, обычно 5432.

`-P` *присваивание*
`--pset=`*присваивание*

Задаёт параметры печати, в стиле команды `\pset`. Обратите внимание, что имя параметра и значение разделяются знаком равенства, а не пробела. Например, чтобы установить формат вывода в LaTeX, нужно написать `-P format=latex`.

`-q`
`--quiet`

Указывает, что psql должен работать без вывода дополнительных сообщений. По умолчанию, выводятся приветствия и различные информационные сообщения. Этого не произойдёт с использованием данного параметра. Полезно вместе с параметром `-c`. Этот же эффект можно получить, установив для переменной `QUIET` значение `on`.

`-R` *разделитель*
`--record-separator=`*разделитель*

Использовать *разделитель* как разделитель записей при невыровненном режиме вывода. Равнозначно команде `\pset recordsep`.

`-s`
`--single-step`

Запуск в пошаговом режиме. Это означает, что пользователь будет подтверждать выполнение каждой команды, отправляемой на сервер, с возможностью отменить выполнение. Используется для отладки скриптов.

`-S`
`--single-line`

Запуск в однострочном режиме, при котором символ новой строки завершает SQL-команды, так же как это делает точка с запятой.

> **Примечание**
>
> Этот режим реализован для тех, кому он нужен, но это не обязательно означает, что и вам нужно его использовать. В частности, если смешивать в одной строке команды SQL и метакоманды, порядок их выполнения может быть не всегда понятен для неопытного пользователя.

`-t`
`--tuples-only`

Отключает вывод имён столбцов и результирующей строки с количеством выбранных записей. Равнозначно команде `\t` или `\pset tuples_only`.

`-T` *`параметры_таблицы`*
`--table-attr=`*`параметры_таблицы`*

Задаёт атрибуты, которые будут вставлены в тег HTML `table`. За подробностями обратитесь к описанию `\pset tableattr`.

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Использовать для подключения к базе данных *`имя_пользователя`* вместо подразумеваемого по умолчанию. (Разумеется, это потребует соответствующего разрешения.)

`-v` *`присваивание`*
`--set=`*`присваивание`*
`--variable=`*`присваивание`*

Выполняет присваивание значения переменной, как метакоманда `\set`. Обратите внимание, что необходимо разделить имя переменной и значение (при наличии) знаком равенства в командной строке. Чтобы сбросить переменную, оставьте имя переменной без знака равенства. Чтобы установить пустое значение, добавьте знак равенства, но опустите значение. Эти присваивания выполняются во время обработки командной строки, так что переменные, отражающие состояние соединения, будут перезаписаны позже.

`-V`
`--version`

Выводит версию psql и завершает работу.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль нельзя получить из других источников, например из файла `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

Обратите внимание, что этот параметр действует на протяжении всей сессии и, таким образом, влияет на метакоманду \connect, так же как и на первую попытку соединения.

-W
--password

Принудительно запрашивать пароль перед подключением к базе данных, даже если он не будет использоваться.

Если сервер требует аутентификацию по паролю и пароль нельзя получить из других источников, например из файла .pgpass, psql запросит пароль в любом случае. Однако чтобы понять, что требуется пароль, psql лишний раз подключится к серверу. Поэтому иногда имеет смысл ввести -W, чтобы исключить эту ненужную попытку подключения.

Обратите внимание, что этот параметр действует на протяжении всей сессии и, таким образом, влияет на метакоманду \connect, так же как и на первую попытку соединения.

-x
--expanded

Включает режим развёрнутого вывода таблицы. Равнозначно команде \x или \pset expanded.

-X,
--no-psqlrc

Не читать стартовые файлы (ни общесистемный файл psqlrc, ни пользовательский файл ~/.psqlrc).

-z
--field-separator-zero

Установить нулевой байт в качестве разделителя полей для невыровненного режима вывода. Равнозначно команде \pset fieldsep_zero.

-0
--record-separator-zero

Установить нулевой байт в качестве разделителя записей для невыровненного режима вывода. Это полезно при взаимодействии с другими программами, например, с xargs -0. Равнозначно команде \pset recordsep_zero.

-1
--single-transaction

Этот параметр может применяться только в сочетании с одним или несколькими параметрами -c и/или -f. С ним psql выполняет команду BEGIN перед обработкой первого такого параметра и COMMIT после последнего, заворачивая таким образом все команды в одну транзакцию. Это гарантирует, что либо все команды завершатся успешно, либо никакие изменения не сохранятся.

Если в самих этих командах содержатся операторы BEGIN, COMMIT или ROLLBACK, этот параметр не даст желаемого эффекта. Кроме того, если какая-либо отдельная команда не может выполняться внутри блока транзакции, с этим параметром вся транзакция прервётся с ошибкой.

-?
--help[=*тема*]

Показать справку по psql и завершиться. Необязательный параметр *тема* (по умолчанию options) выбирает описание интересующей части psql: commands описывает команды psql с обратной косой чертой; options описывает параметры командной строки, которые можно передать psql; а variables выдаёт справку по переменным конфигурации psql.

Код завершения

При нормальном завершении psql возвращает 0 в командную оболочку ОС, 1 — если произошла фатальная ошибка в самом psql (например, нехватка памяти, файл не найден), 2 — при неудачном соединении с сервером неинтерактивного сеанса, 3 — при ошибке в скрипте и установленной переменной `ON_ERROR_STOP`.

Использование

Подключение к базе данных

psql это клиент для Postgres Pro. Для подключения к базе данных нужно знать имя базы данных, имя сервера, номер порта сервера и имя пользователя, под которым вы хотите подключиться. Эти свойства можно задать через аргументы командной строки, а именно `-d`, `-h`, `-p` и `-U` соответственно. Если в командной строке есть аргумент, который не относится к параметрам psql, то он используется в качестве имени базы данных (или имени пользователя, если база данных уже задана). Задавать все эти аргументы необязательно, у них есть разумные значения по умолчанию. Если опустить имя сервера, psql будет подключаться через Unix-сокет к локальному серверу, либо подключаться к `localhost` по TCP/IP в системах, не поддерживающих UNIX-сокеты. Номер порта по умолчанию определяется во время компиляции. Поскольку сервер базы данных использует то же значение по умолчанию, чаще всего указывать номер порта не нужно. Имя пользователя по умолчанию, как и имя базы данных по умолчанию, совпадает с именем пользователя в операционной системе. Заметьте, что просто так подключаться к любой базе данных под любым именем пользователя вы не сможете. Узнать о ваших правах можно у администратора баз данных.

Если значения по умолчанию не подходят, можно сэкономить на вводе параметров подключения, установив переменные среды `PGDATABASE`, `PGHOST`, `PGPORT` и/или `PGUSER`. (Другие переменные среды описаны в Разделе 31.14.) Также удобно иметь файл `~/.pgpass`, чтобы не вводить пароли снова и снова. За дополнительными сведениями обратитесь к Разделу 31.15.

Альтернативный вариант указания параметров подключения — использование строки *conninfo* или URI вместо имени базы данных. Этот механизм даёт широкие возможности для управления соединением. Например:

```
$ psql "service=myservice sslmode=require"
$ psql postgresql://dbmaster:5433/mydb?sslmode=require
```

Этот способ также позволяет использовать LDAP для получения параметров подключения, как описано в Разделе 31.17. Более полно все имеющиеся параметры соединения описаны в Подразделе 31.1.2.

Если соединение не может быть установлено по любой причине (например, нет прав, сервер не работает и т. д.), psql вернёт ошибку и прекратит работу.

Если и стандартный ввод, и стандартный вывод являются терминалом, то psql установит кодировку клиента в «auto», и подходящая клиентская кодировка будет определяться из локальных установок (переменная окружения `LC_CTYPE` в Unix). Если это работает не так, как ожидалось, кодировку клиента можно изменить, установив переменную окружения `PGCLIENTENCODING`.

Ввод SQL-команд

Как правило, приглашение psql состоит из имени базы данных, к которой psql в данный момент подключён, а затем строки `=>`. Например:

```
$ psql testdb
psql (10.23.1)
Type "help" for help.

testdb=>
```

В командной строке пользователь может вводить команды SQL. Обычно введённые строки отправляются на сервер, когда встречается точка с запятой, завершающая команду. Конец

строки не завершает команду. Это позволяет разбивать команды на несколько строк для лучшего понимания. Если команда была отправлена и выполнена без ошибок, то результат команды выводится на экран.

Если к базе данных, которая не приведена в соответствие шаблону безопасного использования схем, имеют доступ недоверенные пользователи, начинайте сеанс с удаления доступных им для записи схем из пути поиска (`search_path`). Для этого можно добавить `options=-csearch_path=` в строку подключения или выполнить команду `SELECT pg_catalog.set_config('search_path', '', false)` перед другими командами SQL. Это касается не только psql, но и любых других интерфейсов для выполнения произвольных SQL-команд.

При каждом выполнении команды psql также проверяет асинхронные уведомления о событиях, генерируемые командами LISTEN и NOTIFY.

Комментарии в стиле C передаются для обработки на сервер, в то время как комментарии в стандарте SQL psql удаляет перед отправкой.

Метакоманды

Всё, что вводится в psql не взятое в кавычки и начинающееся с обратной косой черты, является метакомандой psql и обрабатывается самим psql. Эти команды делают psql полезным для задач администрирования и разработки скриптов.

Формат команды psql следующий: обратная косая черта, сразу за ней команда, затем аргументы. Аргументы отделяются от команды и друг от друга любым количеством пробелов.

Чтобы включить пробел в значение аргумента, нужно заключить его в одинарные кавычки. Чтобы включить одинарную кавычку в значение аргумента, нужно написать две одинарные кавычки внутри текста в одинарных кавычках. Всё, что содержится в одинарных кавычках подлежит заменам, принятым в языке C: `\n` (новая строка), `\t` (табуляция), `\b` (backspace), `\r` (возврат каретки), `\f` (подача страницы), `\`*цифры* (восьмеричное число), и `\x`*цифры* (шестнадцатеричное число). Если внутри текста в одинарных кавычках встречается обратная косая перед любым другим символом, то она экранирует этот символ.

Если внутри аргумента не в кавычках встречается имя переменной psql с предшествующим двоеточием (`:`), оно заменяется значением переменной, как описано в Подразделе «Интерполяция SQL». Также будет работать описанная ниже форма `:'`*имя_переменной*`'` и `:"`*имя_переменной*`"`.

Текст аргумента, заключённый в обратные кавычки (`` ` ``), считается командной строкой, которая передаётся в командную оболочку ОС. Вывод от этой команды (с удалёнными в конце символами новой строки) заменяет текст в обратных кавычках. В содержимом этого текста не обрабатываются никакие спецпоследовательности или особые знаки, за исключением того, что все вхождения `:`*имя_переменной*, где *имя_переменной* — это имя переменной psql, заменяются значением этой переменной. Также вхождения `:'`*имя_переменной*`'` заменяются значением переменной, заключённым в апострофы с тем, что это было одним аргументом команды оболочки. (Последняя форма почти всегда более предпочтительна, если только вы не абсолютно точно знаете, чего ожидать в переменной.) Так как символы перевода строки и возврата каретки могут быть надёжно экранированы не на всех платформах, форма `:'`*имя_переменной*`'` выводит сообщение об ошибке и подстановка значения переменной не производится, когда это значение содержит такие символы.

Некоторые команды принимают идентификатор SQL (например, имя таблицы) в качестве аргумента. Такие аргументы следуют правилам синтаксиса SQL: буквы, не взятые в кавычки, преобразуются в нижний регистр, буквы, взятые в двойные кавычки (") предотвращают преобразование регистра и позволяют включать пробелы в идентификатор. Внутри двойных кавычек две двойные кавычки сокращаются до одной. Например, `FOO"BAR"BAZ` интерпретируется как `fooBARbaz`, а `"A weird"" name"` становится `A weird" name`.

Разбор аргументов останавливается в конце строки или когда встречается другая обратная косая черта, не внутри кавычек. Обратная косая не внутри кавычек рассматривается как начало новой метакоманды. Специальная последовательность `\\` (две обратных косых черты) обозначает

окончание аргументов, далее продолжается разбор команд SQL, если таковые имеются. Таким образом, команды SQL и psql можно свободно смешивать в одной строке. Но в любом случае аргументы метакоманды не могут выходить за пределы текущей строки.

Многие из метакоманд оперируют с *буфером текущего запроса*. Этот буфер содержит произвольный текст команд SQL, который был введён, но ещё не отправлен серверу для выполнения. В него будут входить и предыдущие строки, а также текст, расположенный в строке метакоманды перед ней.

Определены следующие метакоманды:

`\a`

Если текущий режим вывода таблицы невыровненный, то он переключается на выровненный режим. Если текущий режим выровненный, то устанавливается невыровненный. Эта команда поддерживается для обратной совместимости. См. `\pset` для более общего решения.

`\c` или `\connect [ -reuse-previous=on|off ] [ имя_бд [ имя_пользователя ] [ компьютер ] [ порт ] | строка_подключения ]`

Устанавливает новое подключение к серверу Postgres Pro. Параметры подключения можно указывать как позиционно (один или несколько по списку: база данных, пользователь, компьютер и порт), так и передавая аргумент *строка_подключения* (подробнее о строках подключения рассказывается в Подразделе 31.1.1). Если аргументы отсутствуют, новое подключение устанавливается с теми же параметрами, что и предыдущее.

Указание в параметре *имя_бд*, *имя_пользователя*, *компьютер* или *порт* значения - равносильно опущению этого параметра.

Новое подключение может повторно использовать параметры предыдущего — не только имя базы данных, пользователя, компьютер и порт, но и, например, *режим_ssl*. По умолчанию параметры используются повторно при позиционной записи, но не когда задана *строка_подключения*. Это поведение может переопределяться первым аргументом, `-reuse-previous=on` или `-reuse-previous=off`. В случае повторного использования параметров любой параметр, явно не заданный в виде позиционного или в *строке_подключения*, заимствуется из параметров текущего подключения. Исключение составляет параметр *hostaddr* — если он был задан в предыдущем подключении, а в новом в позиционной записи задаётся другое значение *host*, значение *hostaddr* сбрасывается. Кроме того, пароль существующего подключения будет использоваться повторно, только если имя пользователя, узел и порт не меняются. Если какой-либо параметр не указан в этой команде и не используется повторно, действует принятое в libpq значение по умолчанию.

Если новое подключение успешно установлено, предыдущее подключение закрывается. Если попытка подключения не удалась (неверное имя пользователя, доступ запрещён и т. д.), то предыдущее соединение останется активным, только если psql находится в интерактивном режиме. Если скрипт выполняется неинтерактивно, обработка немедленно останавливается с сообщением об ошибке. Различное поведение выбрано для удобства пользователя в качестве защиты от опечаток с одной стороны и в качестве меры безопасности, не позволяющей случайно запустить скрипты в неправильной базе, с другой.

Примеры:

```
=> \c mydb myuser host.dom 6432
=> \c service=foo
=> \c "host=localhost port=5432 dbname=mydb connect_timeout=10 sslmode=disable"
=> \c -reuse-previous=on sslmode=require    -- меняется только sslmode
=> \c postgresql://tom@localhost/mydb?application_name=myapp
```

`\C [ заголовок ]`

Задаёт заголовок, который будет выводиться для результатов любых запросов или отменяет установленный ранее заголовок. Эта команда эквивалентна `\pset title заголовок`. (Название

этой команды происходит от «caption» (заголовок), так как изначально она применялась только для задания заголовков HTML-таблиц.)

`\cd [ *каталог* ]`

Сменяет текущий рабочий каталог на *каталог*. Без аргумента устанавливается домашний каталог текущего пользователя.

> **Подсказка**
>
> для печати текущего рабочего каталога используйте `\! pwd`.

`\conninfo`

Выводит информацию о текущем подключении к базе данных.

`\copy { *таблица* [ ( *список_столбцов* ) ] | ( *запрос* ) } { from | to } { '*имя_файла*' | program '*команда*' | stdin | stdout | pstdin | pstdout } [ [ with ] ( *параметр* [, ...] ) ]`

Производит копирование данных с участием клиента. При этом выполняется SQL-команда COPY, но вместо чтения или записи в файл на сервере, psql читает или записывает файл и пересылает данные между сервером и локальной файловой системой. Это означает, что для доступа к файлам используются привилегии локального пользователя, а не сервера, и не требуются привилегии суперпользователя SQL.

С указанием `program` psql выполняет *команду* и данные, поступающие из/в неё, передаются между сервером и клиентом. Это опять же означает, что для выполнения программ используются привилегии локального пользователя, а не сервера, и не требуются привилегии суперпользователя SQL.

При выполнении `\copy ... from stdin` строки с данными считываются из источника, выполнившего команду, и считываются до тех пор, пока не встретится `\.` или не будет достигнут конец файла. Этот параметр полезен для заполнения таблиц прямо в SQL-скриптах. При выполнении `\copy ... to stdout` вывод направляется в то же место, что и вывод psql команд. Статус команды `COPY *count*` не отображается, чтобы не перепутать со строкой данных. Для чтения/записи стандартного ввода/вывода psql, вне зависимости от источника текущей команды или параметра `\o`, используйте `from pstdin` или `to pstdout`.

Синтаксис команды похож на синтаксис SQL-команды COPY. Все параметры, кроме источника и получателя данных, задаются так же, как и в COPY. Поэтому при обработке метакоманды `\copy` применяются другие правила разбора. В отличие от большинства других метакоманд, для неё остаток строки всегда воспринимается как аргументы `\copy`, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек.

> **Подсказка**
>
> Альтернативный способ получить тот же результат, что и с `\copy ... to` — использовать SQL-команду `COPY ... TO STDOUT` и завершить её командой `\g *имя*` или `\g |*программа*`. В отличие от `\copy`, этот подход позволяет разбивать команду на несколько строк, а также использовать интерполяцию переменных и раскрытие обратных кавычек (`).

> **Подсказка**
>
> Эти операции не так эффективны, как SQL-команда `COPY`, в которой источником или получателем данных является файл или программа, потому что все данные

перемещаются между клиентом и сервером. Для больших объёмов данных SQL-команда может быть предпочтительнее.

`\copyright`

Показывает информацию об авторских правах и условиях распространения Postgres Pro.

`\crosstabview [ столбВ [ столбГ [ столбТ [ столбсортГ ] ] ] ]`

Выполняет содержимое буфера текущего запроса (как `\g`) и показывает результат в виде перекрёстной таблицы. Заданный запрос должен возвращать минимум три столбца. Столбец результата, заданный параметром *столбВ*, будет образовывать вертикальные заголовки, а столбец, заданный параметром *столбГ*, — горизонтальные. Заданный параметром *столбТ* столбец будет поставлять данные для отображения внутри таблицы. Столбец, выбранный параметром *столбсортГ*, будет необязательным столбцом сортировки горизонтальных заголовков.

Каждое указание столбца может представлять собой имя или номер столбца (начиная с 1). К именам применяются обычные принятые в SQL правила учёта регистра и кавычек. По умолчанию в качестве *столбВ* подразумевается столбец 1, а в качестве *столбГ* — столбец 2. Если *столбВ* и *столбГ* задаются явно, они должны различаться. Если *столбТ* не задан, в результате запроса должно быть ровно три столбца, и в качестве *столбТ* выбирается столбец, отличный от *столбВ* и *столбГ*.

Вертикальный заголовок, выводимый в самом левом столбце, содержит значения из столбца *столбВ*, в том же порядке, в каком их возвращает запрос, но без дубликатов.

Горизонтальный заголовок, выводимый в первой строке, содержит значения из столбца *столбГ*, без дубликатов. По умолчанию они располагаются в том порядке, в каком их возвращает запрос. Но если задан необязательный аргумент *столбсортГ*, он определяет столбец, который должен содержать целые числа, и тогда значения из *столбГ* будут располагаться в горизонтальном заголовке по порядку значений в *столбсортГ*.

Внутри перекрёстной таблицы для каждого уникального значения `x` в *столбГ* и каждого уникального значения `y` в *столбВ*, ячейка, размещённая на пересечении `(x,y)` содержит значение `столбТ` в строке результата запроса, в которой значение *столбГ* равно `x`, а значение *столбВ* — `y`. Если такой строки не находится, ячейка остаётся пустой. Если же находится несколько таких строк, выдаётся ошибка.

`\d[S+] [ шаблон ]`

Для каждого отношения (таблицы, представления, материализованного представления, индекса, последовательности, внешней таблицы) или составного типа, соответствующих *шаблону*, показывает все столбцы, их типы, табличное пространство (если оно изменено) и любые специальные атрибуты, такие как `NOT NULL` или значения по умолчанию. Также показываются связанные индексы, ограничения, правила и триггеры. Для сторонних таблиц также показывается связанный сторонний сервер. («Соответствие шаблону» определяется ниже в Подразделе «Шаблоны поиска».)

Для некоторых типов отношений `\d` показывает дополнительную информацию по каждому столбцу: значения столбца для последовательностей, индексируемые выражения для индексов и параметры обёртки сторонних данных для сторонних таблиц.

Вариант команды `\d+` похож на `\d`, но выводит больше информации: комментарии к столбцам таблицы, наличие в таблице OID, для представления показывается его определение, отличные от значений по умолчанию установки replica identity.

По умолчанию отображаются только объекты, созданные пользователем. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`.

> **Примечание**
>
> Если `\d` используется без аргумента *шаблон*, эта команда удобства ради воспринимается как `\dtvmsE` и выдаёт список всех видимых таблиц, представлений, мат. представлений, последовательностей и сторонних таблиц.

`\da[S] [ шаблон ]`

Выводит список агрегатных функций вместе с типом возвращаемого значения и типами данных, которыми они оперируют. Если указан *шаблон*, показываются только те агрегатные функции, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`.

`\dA[+] [ шаблон ]`

Выводит список методов доступа. Если указан *шаблон*, показываются только те методы доступа, имена которых соответствуют ему. При добавлении `+` к команде для каждого метода доступа показывается его описание и связанная функция-обработчик.

`\db[+] [ шаблон ]`

Выводит список табличных пространств. Если указан *шаблон*, показываются только те табличные пространства, имена которых соответствуют ему. При добавлении `+` к команде для каждого объекта дополнительно выводятся параметры, объём на диске, права доступа и описание.

`\dc[S+] [ шаблон ]`

Выводит список преобразований между кодировками наборов символов. Если указан *шаблон*, показываются только те преобразования кодировок, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`. При добавлении `+` к команде для каждого объекта дополнительно будет выводиться описание.

`\dC[+] [ шаблон ]`

Выводит список приведений типов. Если указан *шаблон*, показываются только те приведения типов, имена которых соответствуют ему. При добавлении `+` к команде для каждого объекта дополнительно будет выводиться описание.

`\dd[S] [ шаблон ]`

Показывает описания объектов следующих видов: `ограничение`, `класс операторов`, `семейство операторов`, `правило` и `триггер`. Описания остальных объектов можно посмотреть соответствующими метакомандами для этих типов объектов.

`\dd` показывает описания для объектов, соответствующих *шаблону*, или для доступных объектов указанных типов, если аргументы не заданы. Но в любом случае выводятся только те объекты, которые имеют описание. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`.

Описания объектов создаются SQL-командой COMMENT.

`\dD[S+] [ шаблон ]`

Выводит список доменов. Если указан *шаблон*, показываются только те домены, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`. При

добавлении + к команде для каждого объекта дополнительно будут выводиться права доступа и описание.

\ddp [*шаблон*]

Выводит список прав доступа по умолчанию. Выводится строка для каждой роли (и схемы, если применимо), для которой права доступа по умолчанию отличаются от встроенных. Если указан *шаблон*, выводятся строки только для тех ролей и схем, имена которых соответствуют ему.

Права доступа по умолчанию устанавливаются командой ALTER DEFAULT PRIVILEGES. Смысл отображаемых прав объясняется в описании GRANT.

\dE[S+] [*шаблон*]
\di[S+] [*шаблон*]
\dm[S+] [*шаблон*]
\ds[S+] [*шаблон*]
\dt[S+] [*шаблон*]
\dv[S+] [*шаблон*]

В этой группе команд буквы E, i, m, s, t и v обозначают соответственно: внешнюю таблицу, индекс, материализованное представление, последовательность, таблицу и представление. Можно указывать все или часть этих букв, в произвольном порядке, чтобы получить список объектов этих типов. Например, \dit выводит список индексов и таблиц. При добавлении + к команде для каждого объекта дополнительно будут выводиться физический размер на диске и описание, при наличии. Если указан *шаблон*, выводятся только объекты, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор S.

\des[+] [*шаблон*]

Выводит список сторонних серверов (мнемоника: «external servers»). Если указан *шаблон*, выводятся только те серверы, имена которых соответствуют ему. Если используется форма \des+, то выводится полное описание каждого сервера, включая права доступа, тип, версию, параметры и описание.

\det[+] [*шаблон*]

Выводит список сторонних таблиц (мнемоника: «external tables»). Если указан *шаблон*, выводятся только те записи, имя таблицы или схемы которых соответствуют ему. Если используется форма \det+, то дополнительно выводятся общие параметры и описание сторонней таблицы.

\deu[+] [*шаблон*]

Выводит список сопоставлений пользователей (мнемоника: «external users»). Если указан *шаблон*, выводятся только сопоставления, в которых имена пользователей соответствуют ему. Если используется форма \deu+, то выводится дополнительная информация о каждом сопоставлении пользователей.

> **Внимание**
>
> \deu+ также может отображать имя и пароль удалённого пользователя, поэтому следует позаботиться о том, чтобы не раскрывать их.

\dew[+] [*шаблон*]

Выводит список обёрток сторонних данных (мнемоника: «external wrappers»). Если указан *шаблон*, выводятся только те обёртки сторонних данных, имена которых соответствуют ему. Если используется форма \dew+, то дополнительно выводятся права доступа, параметры и описание обёртки.

`\df[antwS+] [ шаблон ]`

Выводит список функций с типами данных их результатов, аргументов и классификацией: «agg» (агрегатная), «normal» (обычная), «trigger» (триггерная) или «window» (оконная). Чтобы получить функции только определённого вида (видов), добавьте в команду соответствующие буквы `a`, `n`, `t` или `w`. Если задан *шаблон*, показываются только те функции, имена которых соответствуют ему. По умолчанию показываются только функции, созданные пользователями; для включения системных объектов нужно задать шаблон или добавить модификатор `S`. Если используется форма `\df+`, то дополнительно выводятся характеристики каждой функции: изменчивость, допустимость распараллеливания, владелец, классификация по безопасности, права доступа, язык, исходный код и описание.

> **Подсказка**
>
> Чтобы найти функции с аргументами или возвращаемыми значениями определённого типа данных, воспользуйтесь имеющейся в постраничнике возможностью поиска в выводе `\df`.

`\dF[+] [ шаблон ]`

Выводит список конфигураций текстового поиска. Если указан *шаблон*, показываются только те конфигурации, имена которых соответствуют ему. Если используется форма `\dF+`, то выводится полное описание для каждой конфигурации, включая базовый синтаксический анализатор и используемые словари для каждого типа фрагментов.

`\dFd[+] [ шаблон ]`

Выводит список словарей текстового поиска. Если указан *шаблон*, показываются только словари, имена которых соответствуют ему. Если используется форма `\dFd+`, то выводится дополнительная информация о каждом словаре, включая базовый шаблон текстового поиска и параметры инициализации.

`\dFp[+] [ шаблон ]`

Выводит список анализаторов текстового поиска. Если указан *шаблон*, показываются только те анализаторы, имена которых соответствуют ему. Если используется форма `\dFp+`, то выводится полное описание для каждого анализатора, включая базовые функции и список типов фрагментов.

`\dFt[+] [ шаблон ]`

Выводит список шаблонов текстового поиска. Если указан *шаблон*, показываются только шаблоны, имена которых соответствуют ему. Если используется форма `\dFt+`, то выводится дополнительная информация о каждом шаблоне, включая имена основных функций.

`\dg[S+] [ шаблон ]`

Выводит список ролей базы данных. (Так как понятия «пользователи» и «группы» были объединены в «роли», эта команда теперь эквивалентна `\du`.) По умолчанию выводятся только роли, созданные пользователями: чтобы увидеть и системные роли, добавьте модификатор `S`. Если указан *шаблон*, выводятся только те роли, имена которых соответствуют ему. Если используется форма `\dg+`, то выводится дополнительная информация о каждой роли; в настоящее время это комментарий роли.

`\dl`

Это псевдоним для `\lo_list`, показывает список больших объектов.

`\dL[S+] [ шаблон ]`

Выводит список процедурных языков. Если указан *шаблон*, выводятся только те языки, имена которых соответствуют ему. По умолчанию показываются только языки, созданные

пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`. При добавлении `+` к команде для каждого языка дополнительно будут выводиться: обработчик вызова, функция проверки, права доступа и является ли язык системным объектом.

`\dn[S+] [ шаблон ]`

Выводит список схем (пространств имён). Если указан *шаблон*, выводятся только те схемы, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`. При добавлении `+` к команде для каждого объекта дополнительно будут выводиться права доступа и описание, при наличии.

`\do[S+] [ шаблон ]`

Выводит список операторов, их операндов и типы результата. Если указан *шаблон*, выводятся только те операторы, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`. При добавлении `+` к команде для каждого оператора будет выводиться дополнительная информация, сейчас это имя функции, на которой основан оператор.

`\dO[S+] [ шаблон ]`

Выводит список правил сортировки. Если указан *шаблон*, выводятся только те правила, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`. При добавлении `+` к команде для каждого объекта дополнительно будет выводиться описание, при наличии. Обратите внимание, что отображаются только правила сортировки, применимые к кодировке текущей базы данных, поэтому результат команды может отличаться для различных баз данных этой же установки PostgreSQL.

`\dp [ шаблон ]`

Выводит список таблиц, представлений и последовательностей с их правами доступа. Если указан *шаблон*, отображаются только таблицы, представления и последовательности, имена которых соответствуют ему.

Для установки прав доступа используются команды GRANT и REVOKE. Смысл отображаемых прав объясняется в описании GRANT.

`\drds [ шаблон-ролей [ шаблон-баз ] ]`

Выводит список установленных параметров конфигурации. Эти параметры могут быть специфичными для роли, специфичными для базы данных, или обеих. Параметры *шаблон-ролей* и *шаблон-баз* используются для отбора определённых ролей и баз данных, соответственно. Если они опущены, или указано `*`, выводятся все параметры конфигурации, в том числе не относящиеся к ролям или базам данных.

Команды ALTER ROLE и ALTER DATABASE используются для определения параметров конфигурации, специфичных для роли или базы данных.

`\dRp[+] [ шаблон ]`

Выводит список реплицируемых публикаций. Если указан *шаблон*, выводятся только те подписки, имена которых соответствуют ему. При добавлении `+` к команде для каждой публикации показываются также связанные с ней таблицы.

`\dRs[+] [ шаблон ]`

Выводит список подписок на репликацию. Если указан *шаблон*, выводятся только те подписки, имена которых соответствуют ему. При добавлении `+` к команде выводятся дополнительные свойства подписок.

`\dT[S+] [ шаблон ]`

Выводит список типов данных. Если указан *шаблон*, выводятся только те типы, имена которых соответствуют ему. При добавлении + к команде для каждого типа данных дополнительно будет выводиться: внутреннее имя типа, размер, допустимые значения для типа `enum` и права доступа. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`.

`\du[S+] [ шаблон ]`

Выводит список ролей базы данных. (Так как понятия «пользователи» и «группы» были объединены в «роли», эта команда теперь равнозначна `\dg`.) По умолчанию выводятся только роли, созданные пользователями: чтобы увидеть и системные роли, добавьте модификатор `S`. Если указан *шаблон*, выводятся только те роли, имена которых соответствуют ему. Если используется форма `\du+`, то выводится дополнительная информация о каждой роли; в настоящее время это комментарий роли.

`\dx[+] [ шаблон ]`

Выводит список установленных расширений. Если указан *шаблон*, выводятся только расширения, имена которых соответствуют ему. Если используется форма `\dx+`, то для каждого расширения выводятся все принадлежащие ему объекты.

`\dy[+] [ шаблон ]`

Выводит список событийных триггеров. Если указан *шаблон*, выводятся только те событийные триггеры, имена которых соответствуют ему. При добавлении + к команде для каждого объекта дополнительно будет выводиться описание.

`\e` или `\edit [имя_файла] [номер_строки]`

Если указано *имя_файла*, файл открывается для редактирования; после выхода из редактора содержимое файла копируется в буфер текущего запроса. Если *имя_файла* не задано, буфер текущего запроса копируется во временный файл, который затем редактируется тем же образом. Либо, если буфер текущего запроса пуст, во временный файл копируется последний выполненный запрос, и он затем так же редактируется.

Новое содержимое буфера запроса затем разбирается согласно обычным правилам psql, при этом весь буфер обрабатывается как одна строка. Все законченные запросы немедленно выполняются; то есть, если буфер запроса содержит точку с запятой или заканчивается ей, выполняется всё его содержимое до этого знака. Оставшийся текст будет ждать выполнения в буфере запроса; вы можете ввести точку с запятой или `\g`, чтобы передать его, либо `\r`, чтобы сбросить его, очистив буфер запроса. Прочтение буфера как одной строки в основном отражается на метакомандах: всё, что находится в буфере после метакоманды, будет воспринято как аргументы метакоманды, даже если этот текст продолжается на нескольких строках. (Поэтому таким способом нельзя выполнять скрипты с метакомандами. Для таких скриптов используйте `\i`.)

Если указан номер строки, psql будет позиционировать курсор на указанную строку файла или буфера запроса. Обратите внимание, что если указан один аргумент и он числовой, psql предполагает, что это номер строки, а не имя файла.

> **Подсказка**
>
> Как настроить редактор и изменить его поведение, рассказывается в разделе «Переменные окружения».

`\echo текст [ ... ]`

Печатает аргументы в стандартный вывод, разделяя их одним пробелом и добавляя в конце перевод строки. Команда полезна для формирования вывода из скриптов. Например:

```
=> \echo `date`
Tue Oct 26 21:40:57 CEST 1999
```

Если первый аргумент `-n` без кавычек, то перевод строки в конце не ставится.

> **Подсказка**
>
> Если используется команда `\o` для перенаправления вывода запросов, возможно, следует применять команду `\qecho` вместо этой.

`\ef [описание_функции [номер_строки]]`

Эта команда извлекает из базы данных определение заданной функции в форме `CREATE OR REPLACE FUNCTION` и отправляет его на редактирование. Редактирование осуществляется таким же образом, как и для `\edit`. После выхода из редактора изменённая команда будет находиться в буфере запроса. Введите точку с запятой или `\g` для выполнения или `\r` для отмены.

Для функции может быть задано только имя или имя и аргументы, например `foo(integer, text)`. Типы аргументов необходимы, если существует более чем одна функция с тем же именем.

Если функция не указана, для редактирования открывается пустая заготовка `CREATE FUNCTION`.

Если указан номер строки, psql будет позиционировать курсор на указанную строку тела функции. (Обратите внимание, что тело функции обычно не начинается на первой строке файла).

В отличие от большинства других метакоманд весь остаток строки всегда воспринимается как аргументы `\ef`, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек.

> **Подсказка**
>
> Как настроить редактор и изменить его поведение, рассказывается в разделе «Переменные окружения».

`\encoding [ кодировка ]`

Устанавливает кодировку набора символов на клиенте. Без аргумента команда показывает текущую кодировку.

`\errverbose`

Повторяет последнее серверное сообщение об ошибке с максимальным уровнем детализации, как если бы переменная `VERBOSITY` имела значение `verbose`, а `SHOW_CONTEXT` — `always`.

`\ev [имя_представления [номер_строки]]`

Эта команда извлекает и открывает на редактирование определение указанного представления в форме команды `CREATE OR REPLACE VIEW`. Редактирование осуществляется так же, как и с `\edit`. После выхода из редактора изменённая команда остаётся в буфере запроса; введите точку с запятой или `\g`, чтобы выполнить её, либо `\r`, чтобы её сбросить.

Если представление не указано, для редактирования открывается пустая заготовка `CREATE VIEW`.

Если указан номер строки, psql установит курсор на заданную строку в определении представления.

В отличие от большинства других метакоманд весь остаток строки всегда воспринимается как аргументы `\ev`, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек.

`\f [ строка ]`

Устанавливает разделитель полей для невыровненного режима вывода запросов. По умолчанию используется вертикальная черта (`|`). Равнозначно команде `\pset fieldsep`.

`\g [ имя_файла ]`
`\g [ |команда ]`

Отправляет содержимое буфера текущего запроса на сервер для выполнения. Если передаётся аргумент, вывод запроса записывается в указанный файл или передаётся через поток заданной команде оболочки, а не отображается как обычно. Вывод направляется в файл или команду, только если запрос успешно вернул 0 или более строк, но не когда запрос завершился неудачно или выполнялась команда, не возвращающая данные.

Если буфер текущего запроса пуст, вместо этого повторно выполняется предыдущий запрос. За исключением этой особенности, метакоманда `\g` без аргумента по сути равнозначна точке с запятой. Метакоманда `\g` с аргументом является «одноразовой» альтернативой команде `\o`.

Если аргумент начинается с `|`, весь остаток строки воспринимается как *команда*, подлежащая выполнению, в которой не производится ни подстановка переменных, ни раскрытие обратных кавычек. Это продолжение строки просто передаётся оболочке в буквальном виде.

`\gexec`

Отправляет буфер текущего запроса на сервер, а затем обрабатывает содержимое каждого столбца каждой строки результата запроса (если он непустой) как SQL-оператор, то есть исполняет его. Например, следующая команда создаст индексы по каждому столбцу `my_table`:

```
=> SELECT format('create index on my_table(%I)', attname)
-> FROM pg_attribute
-> WHERE attrelid = 'my_table'::regclass AND attnum > 0
-> ORDER BY attnum
-> \gexec
CREATE INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
```

Генерируемые запросы выполняются в том порядке, в каком возвращаются строки, слева направо, если возвращается несколько столбцов. Поля NULL игнорируются. Эти запросы передаются для обработки на сервер буквально, так что это не могут быть метакоманды psql или запросы, использующие переменные psql. В случае сбоя в одном из запросов, выполнение оставшихся запросов продолжается, если только не установлена переменная `ON_ERROR_STOP`. На выполнение каждого запроса оказывает влияние параметр `ECHO`. (Применяя команду `\gexec`, рекомендуется устанавливать в `ECHO` режим `all` или `queries`.) Такие расширенные средства, как протоколирование запросов, пошаговый режим, замер времени и т. п., так же действуют при выполнении каждого генерируемого запроса.

Если буфер текущего запроса пуст, будет повторно выполнен последний переданный запрос.

`\gset [ префикс ]`

Отправляет буфер текущего запроса на сервер для выполнения и сохраняет результат запроса в переменных psql (см. раздел «Переменные»). Выполняемый запрос должен возвращать ровно одну строку. Каждый столбец строки результата сохраняется в отдельной переменной, которая называется так же, как и столбец. Например:

```
=> SELECT 'hello' AS var1, 10 AS var2
```

```
-> \gset
=> \echo :var1 :var2
hello 10
```

Если указан *префикс*, то он добавляется в начале к именам переменных:

```
=> SELECT 'hello' AS var1, 10 AS var2
-> \gset result_
=> \echo :result_var1 :result_var2
hello 10
```

Если значение столбца NULL, то вместо присваивания значения соответствующая переменная удаляется.

Если запрос завершается ошибкой или не возвращает одну строку, то никакие переменные не меняются.

Если буфер текущего запроса пуст, будет повторно выполнен последний переданный запрос.

`\gx [ имя_файла ]`
`\gx [ |команда ]`

Метакоманда `\gx` подобна `\g`, но принудительно включает расширенный режим вывода для текущего запроса. См. `\x`.

`\h` или `\help [ команда ]`

Выводит подсказку по синтаксису указанной команды SQL. Если *команда* не указана, то psql выводит список всех команд, для которых доступна справка. Если в качестве command указана звёздочка (`*`), то выводится справка по всем командам SQL.

В отличие от большинства других метакоманд весь остаток строки всегда воспринимается как аргументы `\help`, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек.

> **Примечание**
>
> Для упрощения ввода команды, состоящие из нескольких слов, можно не заключать в кавычки. Таким образом, можно просто писать **`\help alter table`**.

`\H` или `\html`

Включает вывод запросов в формате HTML. Если формат HTML уже включён, происходит переключение обратно на выровненный формат. Эта команда используется для совместимости и удобства, но в описании `\pset` вы можете узнать о других вариантах вывода.

`\i` или `\include` *имя_файла*

Читает ввод из файла *имя_файла* и выполняет его, как будто он был набран на клавиатуре.

Если *имя_файла* задано как – (минус), читается стандартный ввод до признака конца файла или до метакоманды `\q`. Это может быть полезно для совмещения интерактивного ввода со вводом команд из файлов. Заметьте, что при этом поведение Readline будет применяться, только если оно активно на внешнем уровне.

> **Примечание**
>
> Если вы хотите видеть строки файла на экране по мере их чтения, необходимо установить для переменной `ECHO` значение `all`.

```
\if выражение
\elif выражение
\else
\endif
```

Эта группа команд реализует вложенные условные блоки. Условный блок должен начинаться командой `\if` и заканчиваться `\endif`. Между ними может быть любое количество предложений `\elif`, которые могут быть дополнительно завершаться одним предложением `\else`. Между командами, формирующими блок условия, могут размещаться (и обычно размещаются) обычные запросы и другие типы команд в обратных кавычках.

Команды `\if` и `\elif` считывают свои аргументы и вычисляют их как логические выражения. Если выражение выдаёт `true`, обработка продолжается как обычно; в противном случае входные строки пропускаются до достижения соответствующих команд `\elif`, `\else` или `\endif`. Как только проверка `\if` или `\elif` оказывается успешной, аргументы последующих команд `\elif` в том же блоке не вычисляются, а считаются ложными. Строки, следующие за `\else`, обрабатываются только если ни одна из предыдущих проверок `\if` или `\elif` не была успешной.

В аргументе *выражение* команд `\if` и `\elif` производится подстановка переменных и раскрытие кавычек, как и в аргументе любой другой команды с обратной косой. После этого полученное значение оценивается как значение переменной да/нет. Так что истинным значением будет любое однозначное вхождение без учёта регистра одной из строк (или подстрок): `true`, `false`, `1`, `0`, `on`, `off`, `yes`, `no`. Например, строки `t`, `T` и `tR` все будут восприниматься как `true`.

Если выражения не приводятся к значениям true или false, будет выдано предупреждение, а их результат будет считаться ложным.

Пропускаемые строки разбираются как обычно (в них выявляются запросы и команды с обратной косой), но не передаются серверу, а команды с обратной косой, отличные от условных (`\if`, `\elif`, `\else`, `\endif`), просто игнорируются. В командах условий проверяется только правильность вложенности. Ссылки на переменные в пропускаемых строках не разворачиваются, как не выполняется и раскрытие обратных кавычек.

Все команды с обратной косой в одном условном блоке должны содержаться в одном исходном файле. Если до того, как будут закрыты все локальные блоки `\if`, в основном файле команд будет достигнут конец файла или встретится включение другого файла (команда `\include`), psql выдаст ошибку.

Например:

```
-- проверка существования двух отдельных записей в базе данных и сохранение
-- результатов в двух разных переменных psql
SELECT
    EXISTS(SELECT 1 FROM customer WHERE customer_id = 123) as is_customer,
    EXISTS(SELECT 1 FROM employee WHERE employee_id = 456) as is_employee
\gset
\if :is_customer
    SELECT * FROM customer WHERE customer_id = 123;
\elif :is_employee
    \echo 'is not a customer but is an employee'
    SELECT * FROM employee WHERE employee_id = 456;
\else
    \if yes
        \echo 'not a customer or employee'
    \else
        \echo 'this will never print'
    \endif
\endif
```

\ir или \include_relative *имя_файла*

Команда \ir похожа на \i, но по-разному интерпретирует относительные имена файлов. При выполнении в интерактивном режиме две команды ведут себя одинаково. Однако при вызове из скрипта \ir интерпретирует имена файлов относительно каталога, в котором расположен скрипт, а не текущего рабочего каталога.

\l[+] или \list[+] [*шаблон*]

Выводит список баз данных на сервере и показывает их имена, владельцев, кодировку набора символов и права доступа. Если указан *шаблон*, выводятся только базы данных, имена которых соответствуют ему. При добавлении + к команде также отображаются: размер базы данных, табличное пространство по умолчанию и описание. (Информация о размере доступна только для баз данных, к которым текущий пользователь может подключиться.)

\lo_export *oid_БО имя_файла*

Читает большой объект с заданным *oid_БО* из базы данных и записывает его в файл *имя_файла*. Обратите внимание, что это несколько отличается от серверной функции lo_export, действующей с правами пользователя, от имени которого работает сервер базы данных, и в файловой системе сервера.

> **Подсказка**
>
> Используйте \lo_list для получения OID больших объектов.

\lo_import *имя_файла* [*комментарий*]

Сохраняет файл в большом объекте Postgres Pro. При этом с объектом может быть связан указанный комментарий. Пример:

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

Ответ указывает на то, что большой объект получил OID 152801, который может быть использован для доступа к вновь созданному объекту в будущем. Для удобства чтения рекомендуется всегда связывать объекты с понятными комментариями. OID и комментарии можно посмотреть с помощью команды \lo_list.

Обратите внимание, что это немного отличается от функции сервера lo_import, так как действует от имени локального пользователя в локальной файловой системе, а не пользователя сервера в файловой системе сервера.

\lo_list

Показывает список всех больших объектов Postgres Pro, хранящихся в базе данных, вместе с предоставленными комментариями.

\lo_unlink *oid_БО*

Удаляет большой объект с *oid_БО* из базы данных.

> **Подсказка**
>
> Используйте \lo_list для получения OID больших объектов.

\o или \out [*имя_файла*]
\o или \out [|*команда*]

Результаты запросов будут сохраняться в файле *имя_файла* или перенаправляться команде оболочки (заданной аргументом *команда*). Если аргумент не указан, результаты запросов перенаправляются на стандартный вывод.

Если аргумент начинается с |, весь остаток строки воспринимается как *команда*, подлежащая выполнению, в которой не производится ни подстановка переменных, ни раскрытие обратных кавычек. Это продолжение строки просто передаётся оболочке в буквальном виде.

«Результаты запросов» включают в себя все таблицы, ответы команд, уведомления, полученные от сервера баз данных, а также вывод от метакоманд, обращающихся к базе (таких как \d), но не сообщения об ошибках.

> **Подсказка**
>
> Чтобы вставить текст между результатами запросов, используйте \qecho.

\p или \print

Печатает содержимое буфера текущего запроса в стандартный вывод. Если этот буфер пуст, будет напечатан последний выполненный запрос.

\password [*имя_пользователя*]

Изменяет пароль указанного пользователя (по умолчанию, текущего пользователя). Эта команда запрашивает новый пароль, шифрует и отправляет его на сервер в виде команды ALTER ROLE. Это гарантирует, что новый пароль не отображается в открытом виде в истории команд, журнале сервера или в другом месте.

\prompt [*текст*] *имя*

Предлагает пользователю ввести значение, которое будет присвоено переменной *имя*. Дополнительно можно указать *текст* подсказки. (Если подсказка состоит из нескольких слов, то её текст нужно взять в одинарные кавычки).

По умолчанию, \prompt использует терминал для ввода и вывода. Однако если используется параметр командной строки -f, \prompt использует стандартный ввод и стандартный вывод.

\pset [*параметр* [*значение*]]

Эта команда устанавливает параметры, влияющие на вывод результатов запросов. Указание *параметр* определяет, какой параметр требуется установить. Семантика *значения* зависит от выбранного параметра. Для некоторых параметров отсутствие *значения* означает переключение значения, либо сброс значения, как описано ниже в разделе конкретного параметра. Если такое поведение не упоминается, то пропуск *значения* приводит к отображению текущего значения параметра.

\pset без аргументов выводит текущий статус всех параметров команды.

Имеются следующие параметры:

border

Здесь *значение* должно быть числом. В целом, чем больше это число, тем больше границ и линий будет в таблицах, но детали зависят от формата. В формате HTML заданное значение напрямую отображается в атрибут border=.... Для большинства других форматов имеют смысл только значения 0 (нет границы), 1 (внутренние разделительные линии) и 2 (граница таблицы), а значения больше 2 воспринимаются как border = 2. Форматы latex и latex-longtable дополнительно поддерживают значение 3, добавляющее разделительные линии между строками данных.

columns

Устанавливает максимальную ширину для формата wrapped, а также ограничение по ширине, свыше которого будет требоваться постраничник или произойдёт переключение

в вертикальное отображение при режиме `expanded auto`. При значении 0 (по умолчанию) максимальная ширина управляется переменной среды `COLUMNS` или шириной экрана, если `COLUMNS` не установлена. Кроме того, если `columns` равно нулю, то формат `wrapped` влияет только на вывод на экран. Если `columns` не равно 0, то это также влияет на вывод в файл или в другую команду через канал.

`expanded` (или `x`)

Указанное *значение* допускает варианты `on` или `off`, которые включают или отключают развёрнутый режим, либо `auto`. Если *значение* опущено, команда включает/отключает режим. Когда развёрнутый режим включён, результаты запроса выводятся в две колонки: имя столбца в левой, данные в правой. Этот режим полезен, если данные не помещаются на экране в обычном «горизонтальном» режиме. При выборе `auto` развёрнутый режим используется, когда результат запроса содержит несколько столбцов и по ширине не умещается на экране; в противном случае используется обычный режим. Режим `auto` распространяется только на форматы `aligned` и `wrapped`. С другими форматами он всегда равнозначен отключённому состоянию.

`fieldsep`

Устанавливает разделитель полей для невыровненного режима вывода запросов. Таким образом, можно формировать вывод, в котором значения будут разделены табуляцией или запятыми. Это может быть предпочтительным для использования в других программах. Для установки символа табуляции в качестве разделителя полей введите `\pset fieldsep '\t'`. По умолчанию используется вертикальная черта (`'|'`).

`fieldsep_zero`

Устанавливает разделитель полей для невыровненного режима вывода в нулевой байт.

`footer`

Возможны два варианта *значения*: `on` или `off`, которые включают или отключают вывод результирующей строки с количеством выбранных записей (*n* строк). Если *значение* не указано, то команда переключает текущее значение в `on` или `off`.

`format`

Устанавливает один из следующих форматов вывода: `unaligned`, `aligned`, `wrapped`, `html`, `asciidoc`, `latex` (использует `tabular`), `latex-longtable` или `troff-ms`. Допускается сокращение слова до уникального значения.

В формате `unaligned` все столбцы размещаются на одной строке и отделяются друг от друга текущим разделителем полей. Это полезно для создания вывода, который будут читать другие программы (например, для вывода данных с разделителем Tab или через запятую).

Формат `aligned` это стандартный, удобочитаемый, хорошо отформатированный текстовый вывод. Используется по умолчанию.

Формат `wrapped` похож на `aligned`, но переносит длинные значения столбцов на новые строки, чтобы общий вывод поместился в заданную ширину. Задание ширины вывода описано в параметре `columns`. Обратите внимание, что psql не будет пытаться переносить на новые строки заголовки столбцов. Поэтому формат `wrapped` работает так же, как `aligned` если общая ширина, требуемая для всех заголовков столбцов, превышает установленную максимальную ширину.

Форматы `html`, `asciidoc`, `latex`, `latex-longtable` и `troff-ms` выводят таблицы, которые предназначены для включения в документы с помощью соответствующего языка разметки. Они не являются полноценными документами! Это может быть не критично для HTML, но для LaTeX обязателен документ-контейнер. Формат `latex-longtable` также требует наличия пакетов LaTeX `longtable` и `booktabs`.

linestyle

Задаёт стиль отрисовки линий границы: ascii, old-ascii или unicode. Допускается сокращение слова до уникального значения. (Это значит, что одной буквы будет достаточно.) Значение по умолчанию: ascii. Этот параметр действует только в форматах aligned и wrapped.

Стиль ascii использует обычные символы ASCII. Символы новой строки в данных показываются с использованием символа + в правом поле. Когда при формате wrapped происходит перенос данных на новую строку (без символа новой строки), ставится точка (.) в правом поле первой строки и точка в левом поле следующей строки.

Стиль old-ascii использует обычные символы ASCII в стиле PostgreSQL 8.4 и раньше. Символы новой строки в данных отображаются, используя символ : вместо левого разделителя полей. Когда происходит перенос данных на новую строку без символа новой строки, символ ; используется вместо левого разделителя полей.

Стиль unicode использует символы Юникода для рисования линий. Символы новой строки в данных показываются с использованием символа возврата каретки в правом поле. Когда при формате wrapped происходит перенос данных на новую строку (без символа новой строки), ставится символ многоточия в правом поле первой строки и в левом поле следующей строки.

Когда значение border больше нуля, параметр linestyle также определяет символы, которыми будут рисоваться границы. Обычные символы ASCII работают везде, но символы Юникода смотрятся лучше на терминалах, распознающих их.

null

Устанавливает строку, которая будет напечатана вместо значения null. По умолчанию не печатается ничего, что можно ошибочно принять за пустую строку. Например, можно было бы предпочесть \pset null '(null)'.

numericlocale

Если задаётся *значение*, возможны два варианта: on или off, которые включают или отключают отображение специфичного для локали символа, разделяющего группы цифр левее десятичной точки. Если *значение* не указано, то команда переключает вывод чисел с локализованного на обычный и обратно.

pager

Управляет использованием постраничника для просмотра результатов запросов и справочной информации psql. Если переменная среды PAGER установлена, то данные передаются в указанную программу. В противном случае используется платформозависимая программа по умолчанию (например, more).

Если pager имеет значение off, программа постраничного просмотра (постраничник) не используется. Если pager имеет значение on, эта программа используется при необходимости, т. е. когда вывод на терминал не помещается на экране. Параметр pager также может иметь значение always, при этом постраничник будет использоваться всегда, независимо от того, помещается вывод на экран терминала или нет. Команда \pset pager без указания *значения* переключает варианты on и off.

pager_min_lines

Если в pager_min_lines задаётся число, превышающее высоту страницы, программа постраничного вывода не будет вызываться, пока не наберётся заданное число строк для вывода. Значение по умолчанию — 0.

recordsep

Устанавливает разделитель записей (строк) для невыровненного режима вывода. По умолчанию используется символ новой строки.

`recordsep_zero`

Устанавливает разделитель записей для невыровненного режима вывода в нулевой байт.

`tableattr` (или `T`)

Устанавливает атрибуты, которые будут помещены в тег `table`, при формате вывода HTML. Например, это может быть `cellpadding` или `border`. Заметьте, что, вероятно, не нужно здесь задавать `border`, так как для этого уже есть `\pset border`. Если *значение* не задано, атрибуты таблицы удаляются.

В формате `latex-longtable` этот параметр контролирует пропорциональную ширину каждого столбца, данные которого выровнены по левому краю. Он указывается как список разделённых пробелами значений, например `'0.2 0.2 0.6'`. Для столбцов, которым не хватает значений, используется последнее из заданных.

`title` (или `C`)

Устанавливает заголовок таблицы для любых впоследствии выводимых таблиц. Это можно использовать для задания описательных тегов при формировании вывода. Если *значение* не задано, заголовок таблицы удаляется.

`tuples_only` (или `t`)

Возможны два варианта *значения*: `on` или `off`, которые включают или отключают режим вывода только кортежей. Если *значение* не указано, то команда переключает с режима вывода только кортежей на обычный режим и обратно. Обычный вывод включает в себя дополнительную информацию, такую как заголовки столбцов и различные колонтитулы. В режиме вывода только кортежей отображаются только фактические табличные данные.

`unicode_border_linestyle`

Устанавливает стиль рисования границ для стиля линий `unicode`: `single` (одинарный) или `double` (двойной).

`unicode_column_linestyle`

Устанавливает стиль рисования колонок для стиля линий `unicode`: `single` (одинарный) или `double` (двойной).

`unicode_header_linestyle`

Устанавливает стиль рисования заголовка для стиля линий `unicode`: `single` (одинарный) или `double` (двойной).

Иллюстрацию того, как могут выглядеть различные форматы, можно увидеть в разделе «Примеры».

> **Подсказка**
>
> Для некоторых параметров `\pset` есть короткие команды. См. `\a`, `\C`, `\f`, `\H`, `\t`, `\T` и `\x`.

`\q` или `\quit`

Выход из psql. При использовании в скрипте прекращается только выполнение этого скрипта.

`\qecho` *текст* `[ ... ]`

Эта команда идентична `\echo` за исключением того, что вывод будет записываться в канал вывода запросов, установленный `\o`.

`\r` или `\reset`

Сбрасывает (очищает) буфер запроса.

`\s [ имя_файла ]`

Записывает историю команд psql в файл *имя_файла*. Если *имя_файла* не указано, история команд выводится в стандартный вывод (с использованием постраничника, когда уместно). Этот параметр недоступен, если psql был собран без поддержки Readline.

`\set [ имя [ значение [ ... ] ] ]`

Задаёт для переменной psql *имя* указанное *значение* или, если указано несколько значений, все эти значения, соединённые вместе. Если присутствует только один аргумент, значением переменной становится пустая строка. Для сброса переменной используйте команду `\unset`.

`\set` без аргументов выводит имена и значения всех psql переменных, установленных в настоящее время.

Имена переменных могут содержать буквы, цифры и знаки подчёркивания. Подробнее см. раздел «Переменные» ниже. Имена переменных чувствительны к регистру.

Некоторые переменные отличаются от остальных, тем что управляют поведением psql или устанавливаются автоматически, отражая состояние соединения. Они описаны ниже, в разделе «Переменные».

> **Примечание**
>
> Эта команда не имеет отношения к SQL-команде SET.

`\setenv имя [ значение ]`

Задаёт для переменной среды *имя значение* или, если *значение* не задано, удаляет переменную среды. Пример:

```
testdb=> \setenv PAGER less
testdb=> \setenv LESS -imx4F
```

`\sf[+] описание_функции`

Извлекает из базы данных и выводит определение заданной функции в форме команды `CREATE OR REPLACE FUNCTION`. Определение печатается в текущий канал вывода запросов, установленный `\o`.

Для функции может быть задано только имя или имя и аргументы, например `foo(integer, text)`. Типы аргументов необходимы, если существует более чем одна функция с тем же именем.

При добавлении `+` к команде строки вывода нумеруются, первая строка тела функции получит номер 1.

В отличие от большинства других метакоманд весь остаток строки всегда воспринимается как аргументы `\sf`, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек.

`\sv[+] имя_представления`

Извлекает из базы данных и выводит определение указанного представления в форме команды `CREATE OR REPLACE VIEW`. Определение выводится в текущий канал вывода запросов, установленный `\o`.

При добавлении `+` к команде строки вывода нумеруются, начиная с 1.

В отличие от большинства других метакоманд весь остаток строки всегда воспринимается как аргументы `\sv`, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек.

`\t`

Включает/выключает отображение имён столбцов и результирующей строки с количеством выбранных записей для запросов. Эта команда эквивалентна `\pset tuples_only` и предоставлена для удобства.

`\T` *параметры_таблицы*

Устанавливает атрибуты, которые будут помещены в тег `table` при формате вывода HTML. Эта команда эквивалентна `\pset tableattr` *параметры_таблицы*.

`\timing [` *on* | *off* `]`

С параметром данная команда, в зависимости от него, включает/отключает отображение времени выполнения каждого SQL-оператора. Без параметра она меняет состояние отображения на противоположное. Время выводится в миллисекундах; интервалы больше 1 секунды выводятся в формате минуты:секунды, а при необходимости в вывод также добавляются часы и дни.

`\unset` *имя*

Удаляет psql переменную *имя*.

Большинство переменных, управляющих поведением psql, нельзя сбросить; команда `\unset` для них воспринимается как установка значений по умолчанию. См. раздел Подраздел «Переменные» ниже.

`\w` или `\write` *имя_файла*
`\w` или `\write |`*команда*

Выводит буфер текущего запроса в файл *имя_файла* или через канал в команду оболочки *команда*. Если этот буфер пуст, будет выведен последний выполненный запрос.

Если аргумент начинается с `|`, весь остаток строки воспринимается как *команда*, подлежащая выполнению, в которой не производится ни подстановка переменных, ни раскрытие обратных кавычек. Это продолжение строки просто передаётся оболочке в буквальном виде.

`\watch [` *секунды* `]`

Эта команда многократно выполняет текущий запрос в буфере (как `\g`), пока не будет прервана или не возникнет ошибка. Аргумент задаёт количество секунд ожидания между выполнениями запроса (по умолчанию 2). Результат каждого запроса выводится с заголовком, включающим строку `\pset title` (если она задана), время запуска запроса и интервал задержки.

Если буфер текущего запроса пуст, будет повторно выполнен последний переданный запрос.

`\x [` *on* | *off* | *auto* `]`

Устанавливает или переключает режим развёрнутого вывода таблицы. Это эквивалентно `\pset expanded`.

`\z [` *шаблон* `]`

Выводит список таблиц, представлений и последовательностей с их правами доступа. Если указан *шаблон*, отображаются только таблицы, представления и последовательности, имена которых соответствуют ему.

Это псевдоним для `\dp` («показать права доступа»).

`\! [ команда ]`

Без аргументов запускает подчинённую оболочку; когда эта оболочка завершается, psql продолжает работу. Если добавлен аргумент, запускает команду оболочки *команда*.

В отличие от большинства других метакоманд весь остаток строки всегда воспринимается как аргументы `\!`, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек. Этот текст просто передаётся оболочке в буквальном виде.

`\? [ тема ]`

Показывает справочную информацию. Необязательный параметр *тема* (по умолчанию `commands`) выбирает описание интересующей части psql: `commands` описывает команды psql с обратной косой чертой; `options` описывает параметры командной строки, которые можно передать psql; а `variables` выдаёт справку по переменным конфигурации psql.

Шаблоны поиска

Различные `\d` команды принимают параметр *шаблон* для указания имени (имён) объектов для отображения. В простейшем случае шаблон — это точное имя объекта. Символы внутри шаблона обычно приводятся к нижнему регистру, как и для имён SQL-объектов; к примеру `\dt FOO` выводит таблицу с именем `foo`. Как и для SQL имён, двойные кавычки вокруг шаблона предотвращают перевод в нижний регистр. Для включения символа двойной кавычки в шаблон используются два символа двойных кавычек подряд внутри шаблона в двойных кавычках. Опять же, это соответствует правилам для SQL-идентификаторов. Например `\dt "FOO""BAR"` будет выводить таблицу с именем `FOO"BAR` (но не `foo"bar`). В отличие от обычных правил для SQL-имён, можно взять в двойные кавычки только часть шаблона, например `\dt FOO"FOO"BAR` будет выводить таблицу с именем `fooFOObar`.

Если *шаблон* вообще не указан, команды `\d` выводят все объекты, видимые с текущим путём поиска схем. Это эквивалентно указанию `*` в качестве шаблона. (Объект считается *видимым*, если схема, к которой он относится, находится в пути поиска, и объект с таким же типом и именем в пути поиска ещё не встречался. Это эквивалентно утверждению, что на объект можно ссылаться по имени, без явного указания схемы.) Чтобы увидеть все объекты в базе данных, независимо от видимости, используйте в качестве шаблона `*.*`.

Внутри шаблона `*` обозначает любое количество символов, включая отсутствие символов. `?` соответствует любому одному символу. (Это соответствует шаблонам имён файлов в Unix.) Например, `\dt int*` отображает все таблицы, имена которых начинаются на `int`. Однако внутри двойных кавычек `*` и `?` теряют своё специальное значение и становятся обычными символами.

Шаблон, содержащий точку (`.`), интерпретируется как шаблон имени схемы, за которым следует шаблон имени объекта. Например, `\dt foo*.*bar*` отображает все таблицы, имена которых включают `bar`, и расположенные в схемах, имена которых начинаются с `foo`. Шаблону, не содержащему точку, могут соответствовать только объекты текущей схемы. Опять же, точка внутри двойных кавычек теряет своё специальное значение.

Опытные пользователи могут использовать возможности регулярных выражений, такие как классы символов. Например `[0-9]` соответствует любой цифре. Все специальные символы регулярных выражений работают как описано в Подразделе 9.7.3, за исключением: `.` используется в качестве разделителя, как говорилось выше; `*` соответствует регулярному выражению `.*`; `?` соответствует `.`, а также символ `$`, который не имеет специального значения. При необходимости эти символы можно эмулировать указывая `?` для эмуляции `.`, `(R+|)` для *R*`*`, `(R|)` для *R*`?`. `$` не требуется, как символ регулярного выражения, потому что шаблон должен соответствовать имени целиком, в отличие от обычной интерпретации регулярных выражений (другими словами, `$` автоматически добавляется в шаблон). Используйте `*` в начале и/или в конце, если не хотите, чтобы шаблон закреплялся. Обратите внимание, что внутри двойных кавычек, все специальные символы регулярных выражений теряют своё специальное значение и соответствуют сами себе. Также, специальные символы регулярных выражений не действуют в шаблонах для имён операторов (т. е. в аргументе команды `\do`).

Расширенные возможности

Переменные

psql предоставляет возможности подстановки переменных подобные тем, что используются в командных оболочках Unix. Переменные представляют собой пары имя/значение, где значением может быть любая строка любой длины. Имя должно состоять из букв (включая нелатинские буквы), цифр и знаков подчёркивания.

Чтобы установить переменную, используется метакоманда psql `\set`. Например:

```
testdb=> \set foo bar
```

присваивает переменной `foo` значение `bar`. Чтобы получить значение переменной, нужно поставить двоеточие перед её именем, например:

```
testdb=> \echo :foo
bar
```

Это работает как в обычных SQL-командах, так и в метакомандах; подробности в разделе «Интерполяция SQL» ниже.

При вызове `\set` без второго аргумента переменной присваивается пустая строка. Для сброса (то есть удаления) переменной используйте команду `\unset`. Чтобы посмотреть значения всех переменных, вызовите `\set` без аргументов.

> **Примечание**
>
> На аргументы `\set` распространяются те же правила подстановки, что и для других команд. Таким образом можно создавать интересные ссылки, например `\set :foo 'something'`, получая «мягкие ссылки» в Perl или «переменные переменных» в PHP. К сожалению (или к счастью?), с этими конструкциями нельзя сделать ничего полезного. С другой стороны, `\set bar :foo` является прекрасным способом копирования переменной.

Некоторые переменные обрабатываются в psql особым образом. Они представляют собой определённые параметры, которые могут быть изменены во время выполнения путём присваивания нового значения, а в некоторых переменных содержится изменяемое состояние psql. По соглашению, имена специальных переменных состоят только из заглавных ASCII-букв (и, возможно, цифр и знаков подчёркивания). Для максимальной совместимости в будущем старайтесь не использовать такие имена для собственных переменных.

Переменные, управляющие поведением psql, обычно нельзя сбросить или задать для них недопустимые значения. Команда `\unset` для них допускается, но воспринимается как установка значения по умолчанию. Команда `\set` без второго аргумента воспринимается как присваивание переменной значения `on`, для управляющих переменных, принимающих это значение, и не принимается для других. Также управляющие переменные, принимающие значения `on` и `off`, примут и другие общепринятые написания логических значений, например `true` и `false`.

Специальные переменные:

`AUTOCOMMIT`

При значении `on` (по умолчанию) после каждой успешно выполненной команды выполняется фиксация изменений. Чтобы отложить фиксацию изменений в этом режиме, нужно выполнить SQL-команду `BEGIN` или `START TRANSACTION`. При значении `off` или если переменная не определена, фиксация изменений не происходит до тех пор, пока явно не выполнена команда `COMMIT` или `END`. При значении `off` неявно выполняется `BEGIN` непосредственно перед любой командой, за исключением случаев когда: команда уже в транзакционном блоке; перед самой командой `BEGIN` или другой командой управления транзакциями; перед командой, которая не может выполняться внутри транзакционного блока (например `VACUUM`).

> **Примечание**
>
> Если режим autocommit отключён, необходимо явно откатывать изменения в неуспешных транзакциях, выполняя команду ABORT или ROLLBACK. Также имейте в виду, что при выходе из сессии без фиксации изменений несохранённые изменения будут потеряны.

> **Примечание**
>
> Включённый режим autocommit является традиционным для Postgres Pro, а выключенный режим ближе к спецификации SQL. Если вы предпочитаете отключить режим autocommit, это можно сделать в общесистемном файле psqlrc или в персональном файле ~/.psqlrc.

COMP_KEYWORD_CASE

Определяет, какой регистр букв будет использован при автоматическом завершении ключевых слов SQL. Если установлено в lower или upper, будет использоваться нижний или верхний регистр соответственно. Если установлено в preserve-lower или preserve-upper (по умолчанию), то завершаемое слово будет в том же регистре, что и уже введённое начало слова, но последующие слова, завершаемые полностью, будут в нижнем или верхнем регистре соответственно.

DBNAME

Имя базы данных, к которой вы сейчас подключены. Устанавливается всякий раз при подключении к базе данных (в том числе при старте программы), но эту переменную можно изменить или сбросить.

ECHO

Со значением all все непустые входящие строки выдаются в стандартный вывод по мере их чтения. (Это не относится к строкам, считываемым интерактивно.) Чтобы выбрать такое поведение при запуске программы, добавьте ключ -a. Со значением queries psql выдаёт каждый запрос, отправляемый серверу, в стандартный вывод. Этому значению соответствует ключ -e. Со значением errors в стандартный канал ошибок выдаются только запросы, вызвавшие ошибки. Ему соответствует ключ -b. Со значением none (по умолчанию), никакие запросы не выводятся.

ECHO_HIDDEN

Если эта переменная имеет значение on и метакоманда обращается к базе данных, сначала выводится текст нижележащего запроса. Это помогает изучать внутреннее устройство Postgres Pro и реализовывать похожую функциональность в своих программах. (Чтобы включить такое поведение при запуске программы, воспользуйтесь ключом -E.) Если вы зададите для этой переменной значение noexec, запросы будут просто показываться, но не будут отправляться на сервер и выполняться. Значение по умолчанию — off.

ENCODING

Текущая кодировка символов на стороне клиента. Устанавливается всякий раз при подключении к базе данных (в том числе при старте программы) и при смене кодировки командой \encoding, но эту переменную можно изменить или сбросить.

FETCH_COUNT

Если значение этой переменной — целое число больше нуля, результаты запросов SELECT извлекаются из базы данных и отображаются группами с заданным количеством строк, в отличие от поведения по умолчанию, когда перед отображением результирующий набор накапливается целиком. Это позволяет использовать ограниченный размер памяти независимо

от размера выборки. При включении этой функциональности обычно используются значения от 100 до 1000. Имейте в виду, что запрос может завершиться ошибкой после отображения некоторого количества строк.

> **Подсказка**
>
> Хотя можно использовать любой формат вывода, формат по умолчанию `aligned` как правило выглядит хуже, потому что каждая группа по `FETCH_COUNT` строк форматируется отдельно, что может привести к разной ширине столбцов в разных группах. Остальные форматы вывода работают лучше.

`HISTCONTROL`

Если переменная имеет значение `ignorespace`, строки, начинающиеся с пробела, не сохраняются в истории. Если она имеет значение `ignoredups`, в историю не добавляются строки, которые в ней уже есть. Значение `ignoreboth` объединяет эти два варианта. Со значением `none` (по умолчанию) в истории сохраняются все строки, считываемые в интерактивном режиме.

> **Примечание**
>
> Эта функциональность была бессовестно списана с Bash.

`HISTFILE`

Имя файла, в котором будет сохраняться список истории команд. Если эта переменная не определена, имя файла берётся из переменной окружения `PSQL_HISTORY`. Если и она не задана, используется имя по умолчанию — `~/.psql_history` или `%APPDATA%\postgresql\psql_history` в Windows. Например, если установить:

```
\set HISTFILE ~/.psql_history- :DBNAME
```

в `~/.psqlrc`, psql будет вести отдельный файл истории для каждой базы данных.

> **Примечание**
>
> Эта функциональность была бессовестно списана с Bash.

`HISTSIZE`

Максимальное число команд, которые будут сохраняться в истории команд (по умолчанию 500). Если задано отрицательное значение, ограничение не накладывается.

> **Примечание**
>
> Эта функциональность была бессовестно списана с Bash.

`HOST`

Имя компьютера, где работает сервер базы данных, к которому вы сейчас подключены. Устанавливается всякий раз при подключении к базе данных (в том числе при старте программы), но эту переменную можно изменить или сбросить.

`IGNOREEOF`

Если равно 1 или меньше, символ конца файла (EOF, обычно передаётся сочетанием клавиш **Control+D**) в интерактивном сеансе psql завершит работу приложения. Если значение больше

1, оно определяет, сколько последовательных символов EOF нужно ввести, чтобы завершить интерактивный сеанс. Если значение переменной не является числовым, оно воспринимается как 10. По умолчанию — 0.

> **Примечание**
>
> Эта функциональность была бессовестно списана с Bash.

`LASTOID`

Содержит значение последнего OID, полученного командой `INSERT` или `\lo_import`. Корректное значение переменной гарантируется до тех пор, пока не будет отображён результат следующей SQL-команды.

`ON_ERROR_ROLLBACK`

При значении `on`, если команда в блоке транзакции выдаёт ошибку, ошибка игнорируется и транзакция продолжается. Со значением `interactive` такие ошибки игнорируются только в интерактивных сеансах, но не в скриптах. Со значением `off` (по умолчанию) команда в блоке транзакции, выдающая ошибку, прерывает всю транзакцию. Для реализации режима отката транзакции за вас неявно выполняется команда `SAVEPOINT` непосредственно перед каждой командой в блоке транзакции, а в случае ошибки команды происходит откат к этой точке сохранения.

`ON_ERROR_STOP`

По умолчанию, после возникновения ошибки обработка команд продолжается. Если эта переменная установлена в значение `on`, обработка команд будет немедленно прекращена. В интерактивном режиме psql вернётся в командную строку; иначе psql прекратит работу с кодом возврата 3, чтобы отличить этот случай от фатальных ошибок, для которых используется код возврата 1. В любом случае выполнение всех запущенных скриптов (высокоуровневый скрипт и любые другие, которые он мог запустить) будет немедленно прекращено. Если высокоуровневая командная строка содержит несколько SQL-команд, выполнение завершится на текущей команде.

`PORT`

Содержит порт сервера базы данных, к которому вы сейчас подключены. Устанавливается всякий раз при подключении к базе данных (в том числе при старте программы), но эту переменную можно изменить или сбросить.

`PROMPT1`
`PROMPT2`
`PROMPT3`

Указывают, как должны выглядеть приглашения psql. См. Подраздел «Настройка приглашений».

`QUIET`

Установка значения `on` эквивалента параметру командной строки `-q`. Это, вероятно, не слишком полезно в интерактивном режиме.

`SERVER_VERSION_NAME`
`SERVER_VERSION_NUM`

Номер версии сервера в виде строки, например `9.6.2`, `10.1` или `11beta1`, и в числовом виде, например, `90602` или `100001`. Они устанавливаются при каждом подключении к базе данных (в том числе при запуске программы), но их можно изменить или сбросить.

SHOW_CONTEXT

Этой переменной можно присвоить значения never (никогда), errors (ошибки) или always (всегда), определяющие, когда в сообщениях с сервера будут выводиться поля КОНТЕКСТ. По умолчанию выбран вариант errors (который означает, что контекст будет выводиться в сообщениях об ошибках, но не в предупреждениях и уведомлениях). Этот параметр не действует, когда установлен уровень VERBOSITY terse. (Когда вам потребуется подробная версия только что выданной ошибки, может быть полезна команда \errverbose.)

SINGLELINE

Установка значения on эквивалентна параметру командной строки -S.

SINGLESTEP

Эта переменная эквивалентна параметру командной строки -s.

USER

Содержит имя пользователя базы данных, который сейчас подключён. Устанавливается всякий раз при подключении к базе данных (в том числе при старте программы), но эту переменную можно изменить или сбросить.

VERBOSITY

Этой переменной можно присвоить значения default, verbose или terse для изменения уровня детализации в сообщениях об ошибках. (См. также команду \errverbose, полезную, когда требуется подробная версия только что выданной ошибки.)

VERSION
VERSION_NAME
VERSION_NUM

Эти переменные устанавливаются при запуске программы и отражают версию psql соответственно в виде развёрнутой строки, краткой строки (например, 9.6.2, 10.1 или 11beta1) и числа (например, 90602 или 100001). Их можно изменить или сбросить.

Интерполяция SQL

Ключевой особенностью переменных psql является возможность подставлять («интерполировать») их в команды SQL, так же как и в аргументы метакоманд. Кроме того, psql предоставляет средства для корректного использования кавычек для значений переменных, которые используются как литералы или идентификаторы SQL. Чтобы подставить значение без кавычек, нужно добавить перед именем переменной двоеточие (:). Например:

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

будет запрашивать таблицу my_table. Обратите внимание, что это может быть небезопасным: значение переменной копируется буквально, поэтому оно может содержать непарные кавычки или даже метакоманды. При применении необходимо убедиться, что это имеет смысл.

Когда значение будет использоваться в качестве SQL литерала или идентификатора, безопаснее заключить его в кавычки. Если значение переменной используется как SQL литерал, то после двоеточия нужно написать имя переменной в одинарных кавычках. Если значение переменной используется как SQL идентификатор, то после двоеточия нужно написать имя переменной в двойных кавычках. Эти конструкции корректно работают с кавычками и другими специальными символами, которые могут содержаться в значении переменной. Предыдущий пример более безопасно выглядит так:

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :"foo";
```

Подстановка переменных не будет выполняться, если SQL литералы или идентификаторы заключены в кавычки. Поэтому конструкция `':foo'` не превратится во взятое в кавычки значение переменной (и это было бы небезопасно, если бы работало, так как обработка кавычек внутри значения переменной была бы некорректной).

Один из примеров использования данного механизма — это копирование содержимого файла в столбец таблицы. Сначала загрузим содержимое файла в переменную, затем подставим значение переменной как строку в кавычках:

```
testdb=> \set content `cat my_file.txt`
testdb=> INSERT INTO my_table VALUES (:'content');
```

(Отметим, что это пока не будет работать, если `my_file.txt` содержит байт NUL. psql не поддерживает NUL в значениях переменных.)

Так как двоеточие может легально присутствовать в SQL-командах, попытка подстановки (например для `:name`, `:'name'` или `:"name"`) не выполняется, если переменная не установлена. В любом случае можно экранировать двоеточие с помощью обратной косой черты, чтобы предотвратить подстановку.

Использование двоеточия для переменных является стандартом SQL для встраиваемых языков запросов, таких как ECPG. Использование двоеточия для срезов массивов и приведения типов является расширениями Postgres Pro, что иногда может конфликтовать со стандартным использованием. Использование двоеточия и кавычек для экранирования значения переменной при подстановке в качестве SQL литерала или идентификатора — это расширение psql.

Настройка приглашений

Приглашения, выдаваемые psql, можно настроить по своему вкусу. Три переменные `PROMPT1`, `PROMPT2` и `PROMPT3` содержат строки и спецпоследовательности, задающие внешний вид приглашения. Приглашение 1 (`PROMPT1`) — это обычное приглашение, которое выдаётся, когда psql ожидает ввода новой команды. Приглашение 2 (`PROMPT2`) выдаётся, когда при вводе команды ожидается дополнительные строки, например потому что команда не была завершена точкой с запятой или не закрыты кавычки. Приглашение 3 (`PROMPT3`) выдаётся при выполнении SQL-команды `COPY FROM STDIN`, когда в терминале нужно ввести значение новой строки.

Значения этих переменных выводятся буквально, за исключением случаев, когда в них встречается знак процента (`%`). В зависимости от следующего символа будет подставляться определённый текст. Существуют следующие подстановки:

`%M`

Полное имя компьютера (с именем домена) сервера базы данных или `[local]`, если подключение выполнено через Unix-сокет, либо `[local:`*`/каталог/имя`*`]`, если при компиляции был изменён путь Unix-сокета по умолчанию.

`%m`

Имя компьютера, где работает сервер баз данных, усечённое до первой точки или `[local]`, если подключение выполнено через Unix-сокет.

`%>`

Номер порта, который прослушивает сервер базы данных.

`%n`

Имя пользователя базы данных для текущей сессии. (Это значение может меняться в течение сессии в результате выполнения команды `SET SESSION AUTHORIZATION`.)

`%/`

Имя текущей базы данных.

`%~`

Похоже на `%/`, но выводит тильду `~`, если текущая база данных совпадает с базой данных по умолчанию.

`%#`

Если пользователь текущей сессии является суперпользователем базы данных, то выводит `#`, иначе `>`. (Это значение может меняться в течение сессии в результате выполнения команды `SET SESSION AUTHORIZATION`.)

`%p`

PID обслуживающего процесса для текущего подключения.

`%R`

В приглашении 1 это обычно символ `=`, но `@`, если сеанс находится в неактивной ветви блока условия, либо `^` в однострочном режиме либо `!`, если сеанс не подключён к базе данных (что возможно при ошибке `\connect`). В приглашении 2 `%R` заменяется символом, показывающим, почему psql ожидает дополнительный ввод: `-`, если команда просто ещё не была завершена, но `*`, если не завершён комментарий `/* ... */`; апостроф, если не завершена строка в апострофах; кавычки, если не завершён идентификатор в кавычках; знак доллара, если не завершена строка в долларах; либо `(`, если после открывающей скобки не хватает закрывающей. В приглашении 3 `%R` не выдаёт ничего.

`%x`

Состояние транзакции: пустая строка, если не в транзакционном блоке; `*`, когда в транзакционном блоке; `!`, когда в транзакционном блоке, в котором произошла ошибка и `?`, когда состояние транзакции не определено (например, нет подключения к базе данных).

`%l`

Номер строки в текущем операторе, начиная с `1`.

`%`*`цифры`*

Подставляется символ с указанным восьмеричным кодом.

`%:`*`имя`*`:`

Значение переменной psql *`имя`*. За подробностями обратитесь к разделу «Переменные».

`%`` `*`команда`*` `` `

Подставляется вывод *`команды`*, как и в обычной подстановке с обратными апострофами.

`%[ ... %]`

Приглашения могут содержать управляющие символы терминала, которые, например, изменяют цвет, фон и стиль текста приглашения или изменяют заголовок окна терминала. Для того, чтобы возможности редактирования Readline работали правильно, непечатаемые символы нужно расположить между `%[` и `%]`, чтобы сделать невидимыми. Можно делать несколько таких включений в приглашение. Например:

```
testdb=> \set PROMPT1 '%[%033[1;33;40m%]%n@%/%R%[%033[0m%]%# '
```

выдаст жирное (`1;`), желтое на черном (`33;40`) приглашение для VT100 совместимых цветных терминалов.

Чтобы вставить знак процента, нужно написать `%%`. По умолчанию используются значения `'%/%R %# '` для `PROMPT1` и `PROMPT2` и `'>> '` для `PROMPT3`.

> **Примечание**
>
> Эта функциональность была бессовестно списана с tcsh.

Редактирование командной строки

psql поддерживает библиотеку Readline для удобного редактирования командной строки. История команд автоматически сохраняется при выходе из psql и загружается при запуске. Завершение клавишей `TAB` также поддерживается, хотя логика завершения не претендует на роль анализатора SQL. Запросы, генерируемые завершением по `TAB`, также могут конфликтовать с другими командами SQL, например `SET TRANSACTION ISOLATION LEVEL`. Если по какой-либо причине вам не нравится завершение по клавише `TAB`, его можно отключить в файле `.inputrc` в вашем домашнем каталоге:

```
$if psql
set disable-completion on
$endif
```

(Это возможность не psql, а Readline. Читайте документацию к Readline для дополнительной информации.)

Переменные окружения

`COLUMNS`

Если `\pset columns` равно нулю, управляет шириной формата вывода `wrapped`, а также определяет, нужно ли использовать постраничник и нужно ли переключаться в вертикальный формат в режиме `expanded auto`.

`PAGER`

Если результат запроса не помещается на экране, он пропускается через эту программу. Обычно это `more` или `less`. Значение по умолчанию зависит от платформы. От использования постраничника можно отказаться, очистив переменную `PAGER` или установив соответствующие параметры с помощью команды `\pset`.

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию (см. Раздел 31.14).

`PSQL_EDITOR`
`EDITOR`
`VISUAL`

Редактор, используемый командами `\e`, `\ef` и `\ev`. Эти переменные рассматриваются в этом же порядке; в силу вступает значение, установленное первым.

По умолчанию в Unix-подобных системах используется `vi`, а в Windows — `notepad.exe`.

`PSQL_EDITOR_LINENUMBER_ARG`

Если в командах `\e`, `\ef` или `\ev` указан номер строки, эта переменная задаёт аргумент командной строки, с которым номер строки может быть передан в редактор. Например, для редакторов Emacs и vi это знак плюс. Добавьте в конец значения пробел, если он требуется для отделения имени аргумента от номера строки. Примеры:

```
PSQL_EDITOR_LINENUMBER_ARG='+'
PSQL_EDITOR_LINENUMBER_ARG='--line '
```

Значение по умолчанию + в Unix-подобных системах (соответствует редактору по умолчанию vi и многим другим распространённым редакторам). На платформе Windows нет значения по умолчанию.

PSQL_HISTORY

Альтернативное расположение файла с историей команд. Допускается использование тильды (~).

PSQLRC

Альтернативное расположение пользовательского файла .psqlrc. Допускается использование тильды (~).

SHELL

Команда операционной системы, выполняемая метакомандой \!.

TMPDIR

Каталог для хранения временных файлов. По умолчанию /tmp.

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 31.14).

Файлы

psqlrc и ~/.psqlrc

При запуске без параметра -X программа psql пытается считать и выполнить команды из общесистемного стартового файла (psqlrc), а затем из персонального стартового файла пользователя (~/.psqlrc), после подключения к базе данных, но перед получением обычных команд. Этими файлами можно воспользоваться для настройки клиента и/или сервера, как правило, с помощью команд \set и SET.

Общесистемный стартовый файл называется psqlrc, он будет искаться в каталоге установки «конфигурация системы». Для того чтобы узнать этот каталог, надёжнее всего выполнить команду pg_config --sysconfdir. По умолчанию он расположен в ../etc/ относительно каталога, содержащего исполняемые файлы Postgres Pro. Имя этого каталога можно задать явно через переменную окружения PGSYSCONFDIR.

Персональный стартовый файл пользователя называется .psqlrc, он будет искаться в домашнем каталоге вызывающего пользователя. В Windows, где отсутствует такое понятие, персональный стартовый файл называется %APPDATA%\postgresql\psqlrc.conf. Расположение персонального стартового файла пользователя можно задать явно через переменную окружения PSQLRC.

Оба стартовых файла, общесистемный и персональный, можно привязать к конкретной версии psql. Для этого в конце имени файла нужно добавить номер основного или корректирующего релиза Postgres Pro, например ~/.psqlrc-9.2 или ~/.psqlrc-9.2.5. При наличии нескольких файлов, файл с более детальным номером версии будет иметь предпочтение.

.psql_history

История командной строки хранится в файле ~/.psql_history или %APPDATA%\postgresql\psql_history на Windows.

Расположение файла истории можно задать явно через переменную psql HISTFILE или через переменную окружения PSQL_HISTORY.

Замечания

- psql лучше всего работает с серверами той же или более старой основной версии. Сбой метакоманды наиболее вероятен, если версия сервера новее, чем версия psql. Однако

команды семейства `\d` должны работать с версиями сервера до 7.4, хотя и необязательно с серверами новее, чем сам psql. Общая функциональность запуска SQL-команд и отображения результатов запросов также должна работать на серверах с более новой основной версией, но это не гарантируется во всех случаях.

Если вы хотите, применяя psql, подключаться к нескольким серверам с различными основными версиями, рекомендуется использовать последнюю версию psql. Также можно собрать копии psql от каждой основной версии и использовать ту, которая соответствует версии сервера. Но на практике в этих дополнительных сложностях нет необходимости.

- В Postgres Pro до версии 9.6 параметр `-c` подразумевал `-X` (`--no-psqlrc`); теперь это не так.
- В PostgreSQL до 8.4 программа psql могла принять первый аргумент однобуквенной команды с обратной косой чертой сразу после команды, без промежуточного пробела. Теперь разделительный пробельный символ обязателен.

Замечания для пользователей Windows

psql создан как «консольное приложение». Поскольку в Windows консольные окна используют кодировку символов отличную от той, что используется для остальной системы, нужно проявить особую осторожность при использовании 8-битных символов. Если psql обнаружит проблемную кодовую страницу консоли, он предупредит вас при запуске. Чтобы изменить кодовую страницу консоли, необходимы две вещи:

- Задать кодовую страницу, выполнив **`cmd.exe /c chcp 1251`**. (1251 это кодовая страница для России, замените на ваше значение.) При использовании Cygwin, эту команду можно записать в `/etc/profile`.
- Установите консольный шрифт в `Lucida Console`, потому что растровый шрифт не работает с кодовой страницей ANSI.

По умолчанию psql работает в кодировке UTF-8 и использует для вывода в консоли API Windows Unicode. Чтобы все символы, поддерживаемые вашим консольным шрифтом Windows отображались корректно, необходимо установить для этой консоли кодовую страницу 65001.

Инсталлятор Postgres Pro для Windows устанавливает постраничник `less.exe` с поддержкой UTF-8 и создаёт ярлык, открывающий консольное окно со шрифтом `Lucida Console` и кодовой страницей 65001. Если вы используете другой постраничник, убедитесь в том, что он тоже поддерживает UTF-8.

Вы можете переопределить кодировку, выбранную в psql по умолчанию, установив переменную окружения `PGCLIENTENCODING`.

Примеры

Первый пример показывает, что для ввода одной команды может потребоваться несколько строк. Обратите внимание, как меняется приглашение:

```
testdb=> CREATE TABLE my_table (
testdb(>  first integer not null default 0,
testdb(>  second text)
testdb-> ;
CREATE TABLE
```

Теперь посмотрим на определение таблицы:

```
testdb=> \d my_table
             Таблица "public.my_table"
 Столбец |   Тип   | Правило сортировки | Допустимость NULL | По умолчанию
 first   | integer |                    | not null          |            0
 second  | text    |                    |                   |
```

Теперь изменим приглашение на что-то более интересное:

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

Предположим, что вы внесли данные в таблицу и хотите на них посмотреть:

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-------+--------
     1 | Один
     2 | Два
     3 | Три
     4 | Четыре
(4 строки)
```

Таблицу можно вывести разными способами при помощи команды `\pset`:

```
peter@localhost testdb=> \pset border 2
Установлен стиль границ: 2.
peter@localhost testdb=> SELECT * FROM my_table;
+-------+--------+
| first | second |
+-------+--------+
|     1 | Один   |
|     2 | Два    |
|     3 | Три    |
|     4 | Четыре |
+-------+--------+
(4 строки)

peter@localhost testdb=> \pset border 0
Установлен стиль границ: 0.
peter@localhost testdb=> SELECT * FROM my_table;
first second
----- ------
    1 Один
    2 Два
    3 Три
    4 Четыре
(4 строки)

peter@localhost testdb=> \pset border 1
Установлен стиль границ: 1.
peter@localhost testdb=> \pset format unaligned
Формат вывода: unaligned.
peter@localhost testdb=> \pset fieldsep ','
Разделитель полей: ",".
peter@localhost testdb=> \pset tuples_only
Выводятся только кортежи.
peter@localhost testdb=> SELECT second, first FROM my_table;
Один,1
Два,2
Три,3
Четыре,4
```

Также можно использовать короткие команды:

```
peter@localhost testdb=> \a \t \x
Формат вывода: aligned.
Режим вывода только кортежей выключен.
Расширенный вывод включён.
peter@localhost testdb=> SELECT * FROM my_table;
```

```
-[ RECORD 1 ]-
first  | 1
second | Один
-[ RECORD 2 ]-
first  | 2
second | Два
-[ RECORD 3 ]-
first  | 3
second | Три
-[ RECORD 4 ]-
first  | 4
second | Четыре
```

Когда это уместно, результаты запроса можно просмотреть в виде перекрёстной таблицы с помощью команды `\crosstabview`:

```
testdb=> SELECT first, second, first > 2 AS gt2 FROM my_table;
 first | second | gt2
-------+--------+-----
     1 | one    | f
     2 | two    | f
     3 | three  | t
     4 | four   | t
(4 rows)

testdb=> \crosstabview first second
 first | one | two | three | four
-------+-----+-----+-------+------
     1 | f   |     |       |
     2 |     | f   |       |
     3 |     |     | t     |
     4 |     |     |       | t
(4 rows)
```

Второй пример показывает таблицу умножения, строки в которой отсортированы в обратном числовом порядке, а столбцы — независимо, по возрастанию числовых значений.

```
testdb=> SELECT t1.first as "A", t2.first+100 AS "B", t1.first*(t2.first+100) as "AxB",
testdb(> row_number() over(order by t2.first) AS ord
testdb(> FROM my_table t1 CROSS JOIN my_table t2 ORDER BY 1 DESC
testdb(> \crosstabview "A" "B" "AxB" ord
 A | 101 | 102 | 103 | 104
---+-----+-----+-----+-----
 4 | 404 | 408 | 412 | 416
 3 | 303 | 306 | 309 | 312
 2 | 202 | 204 | 206 | 208
 1 | 101 | 102 | 103 | 104
(4 rows)
```

reindexdb

reindexdb — переиндексировать базу данных Postgres Pro

Синтаксис

`reindexdb` [*параметр-подключения*...] [*параметр*...] [`-S` | `--schema` *схема*] ... [`-t` | `--table` *таблица*] ... [`-i` | `--index` *индекс*] ... [*имя_бд*]

`reindexdb` [*параметр-подключения*...] [*параметр*...] `-a` | `--all`

`reindexdb` [*параметр-подключения*...] [*параметр*...] `-s` | `--system` [*имя_бд*]

Описание

Утилита reindexdb предназначена для перестроения индексов в базе данных Postgres Pro.

Утилита reindexdb представляет собой обёртку SQL-команды REINDEX. Переиндексация базы данных с её помощью по сути не отличается от переиндексации при обращении к серверу другими способами.

Параметры

reindexdb принимает следующие аргументы командной строки:

`-a`
`--all`

Переиндексировать все базы данных.

`[-d]` *имя_бд*
`[--dbname=]`*имя_бд*

Указывает имя базы данных для переиндексации, когда не используется параметр `-a`/`--all`. Если это указание отсутствует, имя базы определяется переменной окружения `PGDATABASE`. Если эта переменная не установлена, именем базы будет имя пользователя, указанное для подключения. В аргументе *имя_бд* может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

`-e`
`--echo`

Выводить команды, которые reindexdb генерирует и передаёт серверу.

`-i` *индекс*
`--index=`*индекс*

Пересоздать только указанный *индекс*. Добавив дополнительные ключи `-i`, можно пересоздать несколько индексов.

`-q`
`--quiet`

Подавлять вывод сообщений о прогрессе выполнения.

`-s`
`--system`

Переиндексировать только системные каталоги базы данных.

`-S` *схема*
`--schema=`*схема*

Переиндексировать только указанную *схему*. Переиндексировать несколько схем можно, добавив несколько ключей `-S`.

`-t` *таблица*
`--table=`*таблица*

Переиндексировать только указанную *таблицу*. Переиндексировать несколько таблиц можно, добавив несколько ключей `-t`.

`-v`
`--verbose`

Вывести подробную информацию во время процесса.

`-V`
`--version`

Сообщить версию reindexdb и завершиться.

`-?`
`--help`

Показать справку по аргументам командной строки reindexdb и завершиться.

Утилита reindexdb также принимает следующие аргументы командной строки в качестве параметров подключения:

`-h` *сервер*
`--host=`*сервер*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *порт*
`--port=`*порт*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

`-U` *имя_пользователя*
`--username=`*имя_пользователя*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как reindexdb запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, reindexdb лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

`--maintenance-db=`*имя_бд*

Указывает имя базы данных, к которой будет выполняться подключение для определения подлежащих переиндексации баз данных, когда используется ключ `-a`/`--all`. Если это имя не указано, будет выбрана база `postgres`, а если она не существует — `template1`. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке. Кроме того, все параметры в строке подключения, за исключением имени базы, будут использоваться и при подключении к другим базам данных.

Переменные окружения

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 31.14).

Диагностика

В случае возникновения трудностей, обратитесь к описаниям REINDEX и psql, где обсуждаются потенциальные проблемы и сообщения об ошибках. Учтите, что на целевом компьютере должен работать сервер баз данных. При этом применяются все свойства подключения по умолчанию и переменные окружения, которые использует клиентская библиотека libpq.

Замечания

Утилите reindexdb может потребоваться подключаться к серверу Postgres Pro несколько раз, и при этом она будет каждый раз запрашивать пароль. В таких случаях удобно иметь файл `~/.pgpass`. За дополнительными сведениями обратитесь к Разделу 31.15.

Примеры

Переиндексирование базы данных `test`:

```
$ reindexdb test
```

Переиндексирование таблицы `foo` и индекса `bar` в базе данных `abcd`:

```
$ reindexdb --table=foo --index=bar abcd
```

См. также

REINDEX

vacuumdb

vacuumdb — выполнить очистку и анализ базы данных Postgres Pro

Синтаксис

`vacuumdb` [*параметр-подключения*...] [*параметр*...] [`-t` | `--table` *таблица* [(*столбец* [,...])]] ... [*имя_бд*]

`vacuumdb` [*параметр-подключения*...] [*параметр*...] `-a` | `--all`

Описание

Утилита vacuumdb предназначена для очистки базы данных Postgres Pro. Кроме того, vacuumdb генерирует внутреннюю статистику, которую использует оптимизатор запросов Postgres Pro.

Утилита vacuumdb представляют собой обёртку SQL-команды VACUUM. Выполнение очистки и анализа баз данных с её помощью по сути не отличается от выполнения тех же действий при обращении к серверу другими способами.

Параметры

Утилита vacuumdb принимает следующие аргументы командной строки:

`-a`
`--all`

Очистить все базы данных.

`[-d] `*`имя_бд`*
`[--dbname=]`*`имя_бд`*

Указывает имя базы данных для очистки или анализа, когда не используется параметр `-a`/`--all`. Если это указание отсутствует, имя базы определяется переменной окружения `PGDATABASE`. Если эта переменная не задана, именем базы будет имя пользователя, указанное для подключения. В аргументе *имя_бд* может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

`-e`
`--echo`

Выводить команды, которые vacuumdb генерирует и передаёт серверу.

`-f`
`--full`

Произвести «полную» очистку.

`-F`
`--freeze`

Агрессивно «замораживать» версии строк.

`-j `*`число_заданий`*
`--jobs=`*`число_заданий`*

Выполнять команды очистки и анализа в параллельном режиме, запуская их одновременно в количестве *число_заданий*. Это сокращает время обработки, но увеличивает нагрузку на сервер.

vacuumdb будет устанавливать несколько подключений к базе данных (в количестве *число_заданий*), так что убедитесь в том, что значение max_connections достаточно велико, чтобы все эти подключения были приняты.

Заметьте, что использование этого режима с параметром `-f` (`FULL`) может привести к отказам из-за взаимоблокировок, если параллельно начнут обрабатываться определённые системные каталоги.

`-q`
`--quiet`

Подавлять вывод сообщений о прогрессе выполнения.

`-t` *таблица* `[ (`*столбец* `[,...]) ]`
`--table=`*таблица* `[ (`*столбец* `[,...]) ]`

Производить очистку или анализ только указанной *таблицы*. Имена столбцов можно указать только в сочетании с параметрами `--analyze` и `--analyze-only`. Добавив дополнительные ключи `-t`, можно обработать несколько таблиц.

> **Подсказка**
>
> Если вы указываете столбцы, вам, вероятно, придётся экранировать скобки в оболочке. (См. примеры ниже.)

`-v`
`--verbose`

Вывести подробную информацию во время процесса.

`-V`
`--version`

Сообщить версию vacuumdb и завершиться.

`-z`
`--analyze`

Также вычислить статистику для оптимизатора.

`-Z`
`--analyze-only`

Только вычислить статистику для оптимизатора (не производить очистку).

`--analyze-in-stages`

Только вычислить статистику для оптимизатора (без очистки), подобно `--analyze-only`. Но для скорейшего получения полезной статистики, выполнить анализ в несколько проходов (в настоящее время, три) с разными параметрами.

Этот параметр полезен при необходимости провести анализ базы данных, только что наполненной данными из архива или командой `pg_upgrade`. С этим параметром vacuumdb постарается получить некоторую статистику как можно скорее, чтобы базой можно было пользоваться, а на следующих проходах вычислит полную статистику.

`-?`
`--help`

Показать справку по аргументам командной строки vacuumdb и завершиться.

Утилита vacuumdb также принимает следующие аргументы командной строки в качестве параметров подключения:

`-h` *сервер*
`--host=`*сервер*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *порт*
`--port=`*порт*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

`-U` *имя_пользователя*
`--username=`*имя_пользователя*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как vacuumdb запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, vacuumdb лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

`--maintenance-db=`*имя_бд*

Указывает имя базы данных, к которой будет выполняться подключение для определения подлежащих очистке баз данных, когда используется ключ `-a`/`--all`. Если это имя не указано, будет выбрана база `postgres`, а если она не существует — `template1`. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке. Кроме того, все параметры в строке подключения, за исключением имени базы, будут использоваться и при подключении к другим базам данных.

Переменные окружения

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 31.14).

Диагностика

В случае возникновения трудностей, обратитесь к описаниям VACUUM и psql, где обсуждаются потенциальные проблемы и сообщения об ошибках. Учтите, что на целевом компьютере должен работать сервер баз данных. При этом применяются все свойства подключения по умолчанию и переменные окружения, которые использует клиентская библиотека libpq.

Замечания

Утилите vacuumdb может потребоваться подключаться к серверу Postgres Pro несколько раз, и при этом она будет каждый раз запрашивать пароль. В таких случаях удобно иметь файл `~/.pgpass`. За дополнительными сведениями обратитесь к Разделу 31.15.

Примеры

Очистка базы данных `test`:

```
$ vacuumdb test
```

Очистка и анализ для оптимизатора базы данных `bigdb`:

```
$ vacuumdb --analyze bigdb
```

Очистка одной таблицы `foo` в базе данных `xyzzy` и анализ только столбца `bar` таблицы для оптимизатора:

```
$ vacuumdb --analyze --verbose --table='foo(bar)' xyzzy
```

См. также

VACUUM

Серверные приложения Postgres Pro

В этой части содержится справочная информация о серверных приложениях и вспомогательных утилитах Postgres Pro. Описываемые команды могут быть полезны только на том компьютере, где работает сервер баз данных. Другие утилиты рассмотрены в Справке: «Клиентские приложения Postgres Pro».

initdb

initdb — создать кластер баз данных Postgres Pro

Синтаксис

initdb [*параметр*...] [--pgdata | -D]*каталог*

Описание

Команда initdb создаёт новый кластер баз данных Postgres Pro. Кластер — это коллекция баз данных под управлением единого экземпляра сервера.

Инициализация кластера базы данных заключается в создании каталогов для хранения данных, формировании общих системных таблиц (относящихся ко всему кластеру, а не к какой-либо базе) и создании баз данных template1 и postgres. Впоследствии все новые базы создаются на основе шаблона template1 (все дополнения, установленные в template1 автоматически копируются в каждую новую базу данных). База postgres используется пользователями, утилитами и сторонними приложениями по умолчанию.

При попытке создать каталог для хранения данных initdb может столкнуться с нехваткой прав доступа, если этот каталог принадлежит суперпользователю root. В таком случае необходимо назначить пользователя базы данных владельцем этого каталога при помощи chown. Затем выполнить su для смены пользователя и дальнейшего выполнения initdb.

Команда initdb должна выполняться от имени пользователя, под которым будет запускаться сервер, так как ему необходим полный доступ к файлам и каталогам, создаваемым initdb. Сервер не может запускаться от имени суперпользователя, поэтому выполнение команды initdb от его лица будет отклонено.

initdb инициализирует локали и кодировки баз данных кластера, которые будут использоваться по умолчанию. Кодировка, порядок сортировки (LC_COLLATE), классы наборов символов (LC_CTYPE, например, заглавные, строчные буквы, цифры) могут устанавливаться раздельно при создании новой базы данных. initdb определяет параметры локали для шаблона template1, которые будут применяться по умолчанию для новых баз.

Чтобы изменить порядок сортировки по умолчанию или классы наборов символов, используются параметры --lc-collate и --lc-ctype. Порядок сортировки, отличающийся от C или POSIX, оказывает влияние на производительность. Поэтому необходимо тщательно выбирать необходимую и достаточную локаль при выполнении initdb.

Другие категории локали можно изменить и после старта сервера. Также можно использовать параметр --locale, чтобы задать локаль для всех категорий одновременно, включая порядок сортировки и классы наборов символов. Значения локалей сервера (lc_*) можно вывести командой SHOW ALL. Узнать об этом больше можно в Разделе 22.1.

Для изменения кодировки по умолчанию используется параметр --encoding. Узнать об этом больше можно в Разделе 22.3.

Параметры

-A *authmethod*
--auth=*authmethod*

Параметр определяет метод аутентификации по умолчанию для локальных пользователей, используемый в файле pg_hba.conf (строки host и local). Программа initdb предварительно внесёт указанный метод аутентификации в pg_hba.conf в записи как обычных соединений, так и соединений репликации.

Не используйте `trust`, если не можете доверять всем локальным пользователям в вашей системе. Режим `trust` используется по умолчанию для облегчения процесса установки.

`--auth-host=`*`authmethod`*

Параметр указывает метод аутентификации для локальных пользователей, подключающихся по TCP/IP, используемый в `pg_hba.conf` (строки `host`).

`--auth-local=`*`authmethod`*

Параметр выбирает метод аутентификации локальных пользователей, подключающихся через Unix-сокет, используемый в `pg_hba.conf` (строки `local`).

`-D` *`каталог`*
`--pgdata=`*`каталог`*

Параметр указывает каталог хранения данных кластера. Это единственный обязательный параметр для команды `initdb`. При этом его можно указать в переменной окружения `PGDATA`, что будет удобным при дальнейшем использовании (`postgres` обращается к этой же переменной).

`-E` *`кодировка`*
`--encoding=`*`кодировка`*

Устанавливает кодировку шаблона и новых баз данных по умолчанию, если не указать иное при их создании. По умолчанию устанавливается исходя из указанной локали, и далее, если не удалось определить, выбирается `SQL_ASCII`. Кодировки, поддерживаемые сервером Postgres Pro, описаны в Подразделе 22.3.1.

`-k`
`--data-checksums`

Применять контрольные суммы на страницах данных для выявления сбоев при вводе/выводе, которые иначе останутся незамеченными. Расчёт контрольных сумм может повлечь заметное снижение производительности. Этот режим можно включить только при инициализации и нельзя изменить позже. Когда контрольные суммы включены, они рассчитываются для всех объектов и во всех базах данных.

По умолчанию кластеры Postgres Pro инициализируются с включёнными контрольными суммами. Чтобы изменить это поведение, добавьте параметр `--no-data-checksums`.

`--no-data-checksums`

Отключить контрольные суммы в страницах данных.

По умолчанию кластеры Postgres Pro инициализируются с включёнными контрольными суммами.

`--locale=`*`локаль`*`[@`*`провайдер`*`]`

Устанавливает локаль кластера по умолчанию. Если флаг не указан, локаль устанавливается согласно окружению, в котором исполняется команда `initdb`. Поддерживаемые локали описаны в Разделе 22.1.

Кроме того, вы можете задать провайдер основного правила сортировки после символа `@`. Допустимые варианты этого указания: `icu` и `libc`. За подробностями обратитесь к Подразделу 22.2.2.

`--lc-collate=`*`локаль`*`[@`*`провайдер`*`]`
`--lc-ctype=`*`локаль`*
`--lc-messages=`*`локаль`*
`--lc-monetary=`*`локаль`*
`--lc-numeric=`*`локаль`*
`--lc-time=`*`локаль`*

Аналогично `--locale` устанавливает необходимую локаль, но в заданной категории.

`--no-locale`

Аналогично флагу `--locale=C`.

`-N`
`--no-sync`

По умолчанию `initdb` ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром `initdb` завершается быстрее, без ожидания, но в случае неожиданного сбоя операционной системы каталог данных может оказаться испорченным. Этот параметр может быть полезен при тестировании; в производственной среде применять его не следует.

`--pwfile=`*`имя_файла`*

Принуждает `initdb` читать пароль суперпользователя базы данных из файла, первая строка которого используется в качестве пароля.

`-S`
`--sync-only`

Безопасно записывает все файлы базы на диск и останавливается. Другие операции initdb при этом не выполняются.

`-T` *`конфигурация`*
`--text-search-config=`*`конфигурация`*

Устанавливает конфигурацию текстового поиска по умолчанию. За дополнительными сведениями обратитесь к default_text_search_config.

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Устанавливает имя суперпользователя базы данных. По умолчанию используется имя пользователя ОС, запустившего `initdb`. По факту, само по себе имя суперпользователя базы данных не важно, но этот параметр позволяет оставить привычное postgres, если имя пользователя ОС другое.

`-W`
`--pwprompt`

Указывает `initdb` запросить пароль, который будет назначен суперпользователю базы данных. Это не важно, если не планируется использовать аутентификацию по паролю. В ином случае этот режим аутентификации оказывается неприменимым, пока пароль не задан.

`-X` *`каталог`*
`--waldir=`*`каталог`*

Этот параметр указывает каталог для хранения журнала предзаписи.

Другие реже используемые параметры описаны здесь:

`-d`
`--debug`

Выводит отладочные сообщения загрузчика и ряд других сообщений, не очень интересных широкой публике. Загрузчик — это приложение `initdb`, используемое для создания каталога таблиц. С этим параметром выдаётся очень много крайне скучных сообщений.

`-L` *`каталог`*

Указывает `initdb`, где необходимо искать входные файлы для развёртывания кластера. Обычно это не требуется. Приложение само запросит эти данные, если будет необходимо.

`-n`
`--no-clean`

По умолчанию, при выявлении ошибки на этапе развёртывания кластера, `initdb` удаляет все файлы, которые к тому моменту были созданы. Параметр предотвращает очистку файлов для целей отладки.

Прочие параметры:

`-V`
`--version`

Выводит версию initdb и останавливается.

`-?`
`--help`

Показывает помощь по аргументам команды initdb и останавливается.

Переменные окружения

`PGDATA`

Указывает каталог хранения данных кластера, можно изменить параметром `-D`.

`TZ`

Указывает часовой пояс кластера по умолчанию. Значение — это полное имя часового пояса (см. Подраздел 8.5.3).

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 31.14).

Замечания

`initdb` можно выполнить командой `pg_ctl initdb`.

См. также

pg_ctl, postgres

pg_archivecleanup

pg_archivecleanup — вычистить файлы архивов WAL Postgres Pro

Синтаксис

`pg_archivecleanup` [*параметр*...] *`расположение_архива старейший_сохраняемый_файл`*

Описание

Утилита pg_archivecleanup предназначена для использования в качестве `archive_cleanup_command` для удаления старых файлов WAL на резервном сервере (см. Раздел 25.2). pg_archivecleanup можно использовать и как отдельную программу для выполнения тех же действий.

Чтобы использовать pg_archivecleanup на резервном сервере, добавьте эту строку в файл конфигурации `recovery.conf`:

```
archive_cleanup_command = 'pg_archivecleanup расположение_архива %r'
```

Здесь *`расположение_архива`* определяет каталог, из которого должны удаляться файлы сегментов WAL.

Вызываемая в качестве archive_cleanup_command, эта программа просматривает *`расположение_архива`* и удаляет все файлы WAL, логически предшествующие значению аргумента `%r`. Целью этой операции является сокращение числа сохраняемых файлов без потери возможности восстановления при перезапуске. Такой вариант использования уместен, когда *`расположение_архива`* указывает на область рабочих файлов конкретного резервного сервера, но *не* когда *`расположение_архива`* — каталог с архивом WAL для долговременного хранения, или когда несколько резервных серверов восстанавливают записи WAL из одного расположения.

При отдельном использовании этой программы из каталога *`расположение_архива`* будут удалены все файлы WAL, логически предшествующие файлу *`старейший_сохраняемый_файл`*. В этом режиме, если вы укажете имя файла с расширением `.partial` или `.backup`, *`старейший_сохраняемый_файл`* будет определяться по имени без расширения. Благодаря такой интерпретации расширения `.backup` будут корректно удалены все файлы WAL, заархивированные до определённой базовой копии. Например, следующая команда удалит все файлы старее файла WAL с именем `000000010000003700000010`:

```
pg_archivecleanup -d archive 000000010000003700000010.00000020.backup

pg_archivecleanup:  keep WAL file "archive/000000010000003700000010" and later
pg_archivecleanup:  removing file "archive/00000001000000370000000F"
pg_archivecleanup:  removing file "archive/00000001000000370000000E"
```

pg_archivecleanup рассчитывает на то, что *`расположение_архива`* доступно для чтения и записи пользователю, владеющему серверным процессом.

Параметры

pg_archivecleanup принимает следующие аргументы командной строки:

`-d`

Выводить подробные отладочные сообщения в `stderr`.

`-n`

Вывести имена файлов, которые должны быть удалены, в `stdout` (не выполняя удаление).

`-V`
`--version`

Вывести версию pg_archivecleanup и завершиться.

`-x` *расширение*

Установить расширение, которое будет убрано из имён файлов до принятия решения об удалении определённых файлов. Это обычно полезно для очистки файлов архивов, которые сжимаются для хранения и получают расширение, задаваемое программой сжатия. Например: `-x .gz`.

`-?`
`--help`

Вывести справку об аргументах командной строки pg_archivecleanup и завершиться.

Замечания

Программа pg_archivecleanup рассчитана на работу с PostgreSQL 8.0 и новее как отдельная утилита, а также с PostgreSQL 9.0 и новее как команда очистки архива.

Программа pg_archivecleanup написана на C; её исходный код легко поддаётся модификации (он содержит секции, предназначенные для изменения при надобности)

Примеры

В системах Linux или Unix можно использовать команду:

```
archive_cleanup_command = 'pg_archivecleanup -d /mnt/standby/archive %r 2>>cleanup.log'
```

Предполагается, что каталог архива физически располагается на резервном сервере, так что команда `archive_command` обращается к нему по NFS, но для резервного сервера эти файлы локальные. Эта команда будет:

- выводить отладочную информацию в `cleanup.log`
- удалять ставшие ненужными файлы из каталога архива

См. также

pg_standby

pg_controldata

pg_controldata — вывести управляющую информацию кластера баз данных Postgres Pro

Синтаксис

`pg_controldata` [*параметр*] [[`-D`] *datadir*]

Описание

`pg_controldata` показывает свойства, установленные командой `initdb`, например, версию каталога. Она также выводит сведения о журнале предзаписи, включая данные контрольной точки. Эта информация относится ко всему кластеру, а не к отдельной базе данных.

Утилита запускается от лица пользователя, создавшего кластер, так как требует права на чтение в каталоге хранения данных. Можно указать путь к каталогу из командной строки, либо использовать значение переменной окружения `PGDATA`. Также поддерживаются флаги `-V` и `--version`, которые выводят версию pg_controldata и прерывают выполнение. Флаги `-?` и `--help` отображают помощь по поддерживаемым командой аргументам.

Переменные окружения

`PGDATA`

Каталог размещения данных кластера по умолчанию

См. также

pgpro_controldata

pg_ctl

pg_ctl — инициализировать, запустить, остановить или управлять сервером Postgres Pro

Синтаксис

`pg_ctl init[db]` [`-D` *каталог_данных*] [`-s`] [`-o` *параметры-initdb*]

`pg_ctl start` [`-D` *каталог_данных*] [`-l` *имя_файла*] [`-W`] [`-t` *секунды*] [`-s`] [`-o` *параметры*] [`-p` *путь*] [`-c`]

`pg_ctl stop` [`-D` *каталог_данных*] [`-m` `s[mart]` | `f[ast]` | `i[mmediate]`] [`-W`] [`-t` *секунды*] [`-s`]

`pg_ctl restart` [`-D` *каталог_данных*] [`-m` `s[mart]` | `f[ast]` | `i[mmediate]`] [`-W`] [`-t` *секунды*] [`-s`] [`-o` *параметры*] [`-c`]

`pg_ctl reload` [`-D` *каталог_данных*] [`-s`]

`pg_ctl status` [`-D` *каталог_данных*]

`pg_ctl promote` [`-D` *каталог_данных*] [`-W`] [`-t` *секунды*] [`-s`]

`pg_ctl kill` *имя_сигнала ид_процесса*

`pg_ctl logrotate` [`-D` *каталог_данных*]

В системах Microsoft Windows также:

`pg_ctl register` [`-D` *каталог_данных*] [`-N` *имя_службы*] [`-U` *имя_пользователя*] [`-P` *пароль*] [`-S` `a[uto]` | `d[emand]`] [`-e` *source*] [`-W`] [`-t` *секунды*] [`-s`] [`-o` *параметры*]

`pg_ctl unregister` [`-N` *имя_службы*]

Описание

pg_ctl — это утилита для начальной инициализации, запуска, остановки, повторного запуска и управления кластером баз данных Postgres Pro (postgres). Сервер можно стартовать в ручном режиме, но pg_ctl реализует задачи направления вывода в журнал и отсоединения от терминала и группы процессов, а также предоставляет удобный интерфейс остановки кластера.

Команда `init` (`initdb`) создаёт кластер баз данных Postgres Pro, то есть коллекцию баз данных, которой будет управлять один экземпляр сервера. Эта команда вызывает программу `initdb`. За подробностями обратитесь к initdb.

Команда `start` запускает сервер. Процесс запускается в фоне, а стандартный ввод связывается с `/dev/null` (или `nul` в Windows). По умолчанию в Unix-подобных системах вывод и ошибки сервера пишутся в устройство стандартного вывода (не ошибок) pg_ctl. Вывод pg_ctl следует перенаправить в файл или процесс, например, приложение ротации журналов rotatelogs; иначе `postgres` будет писать вывод в управляющий терминал (в фоновом режиме) и останется в группе процессов оболочки. В Windows сообщения и ошибки сервера по умолчанию перенаправляются в терминал. Это поведение по умолчанию можно изменить и направить вывод сервера в файл, добавив ключ `-l`. Предпочтительными вариантами является использование `-l` или перенаправление вывода.

Команда `stop` останавливает сервер, работающий с указанным каталогом данных. Параметр `-m` позволяет выбрать один из трёх режимов остановки. Режим «Smart» запрещает новые подключения, а затем ожидает отключения всех существующих клиентов и завершения всех текущих процессов резервного копирования. Если сервер работает в режиме горячего резерва, восстановление и потоковая репликация будут прерваны, как только отключатся все клиенты. Режим «Fast» (выбираемый по умолчанию) не ожидает отключения клиентов и завершает

все текущие процессы резервного копирования. Все активные транзакции откатываются, а клиенты принудительно отключаются, после чего сервер останавливается. Режим «Immediate» незамедлительно прерывает все серверные процессы, не выполняя процедуру штатной остановки. Этот вариант влечёт необходимость выполнить восстановление после сбоя при следующем запуске сервера.

Команда `restart` по сути производит остановку и последующий запуск сервера. Это позволяет изменить параметры командной строки `postgres` либо применить изменения в файле конфигурации, не вступающие в силу без перезапуска сервера. Если в командной строке при запуске сервера указывались относительные пути, команда `restart` может не выполниться, если вызвать pg_ctl не в том каталоге, где производился предыдущий запуск.

Команда `reload` просто посылает процессу сервера `postgres` сигнал SIGHUP, получив который он перечитывает свои файлы конфигурации (`postgresql.conf`, `pg_hba.conf` и т. д.). Это позволяет применить изменения параметров в файле конфигурации, не требующие полного перезапуска сервера.

Команда `status` проверяет, работает ли сервер в указанном каталоге данных. Если да, она выдаёт PID сервера и параметры командной строки, с которыми он был запущен. Если сервер не работает, pg_ctl возвращает код завершения 3. Если в параметрах не указан доступный каталог данных, pg_ctl возвращает код завершения 4.

Команда `promote` указывает серверу, работающему в режиме резерва с указанным каталогом данных, выйти из этого режима и начать операции чтения/записи.

Команда `kill` передаёт сигнал заданному процессу. Прежде все это полезно в Microsoft Windows, где отсутствует встроенная команда kill. Для получения списка имён поддерживаемых сигналов воспользуйтесь ключом `--help`.

Команда `logrotate` прокручивает файл журнала сервера. Подробнее о том, как использовать это с внешними средствами прокрутки журнала, рассказывается в Разделе 23.3.

Команда `register` регистрирует сервер Postgres Pro в качестве системной службы в Microsoft Windows. Параметр `-S` позволяет выбрать тип запуска службы: «auto» (запускать службу автоматически при загрузке системы) или «demand» (запускать службу по требованию).

Режим `unregister` разрегистрирует системную службу в Microsoft Windows. Эта операция отменяет действие команды `register`.

Параметры

`-c`
`--core-files`

Способствует сбросу дампа памяти процесса при крахе сервера на платформах, где это возможно, поднимая мягкие ограничения, задаваемые для файлов дампа. Это полезно при отладке и диагностике проблем, так как позволяет получить трассировку стека отказавшего процесса сервера.

`-D` *`каталог_данных`*
`--pgdata=`*`каталог_данных`*

Указывает размещение конфигурационных файлов кластера. Если этот ключ опущен, используется значение переменной окружения `PGDATA`.

`-l` *`имя_файла`*
`--log=`*`имя_файла`*

Направляет вывод сообщений сервера в файл *`имя_файла`*. Файл создаётся, если он ещё не существует. При этом устанавливается umask 077, что предотвращает доступ других пользователей к этому файлу.

`-m` *режим*
`--mode=`*режим*

Задаёт режим остановки кластера. Значением *режим* может быть `smart`, `fast` или `immediate`, либо первая буква этих вариантов. Если этот ключ опущен, по умолчанию выбирается режим `fast`.

`-o` *параметры*
`--options=`*параметры*

Указывает параметры, которые будут передаваться непосредственно программе `postgres`. Ключ `-o` можно указывать несколько раз, при этом ей будут переданы параметры из всех ключей.

Задаваемые *параметры* обычно следует обрамлять одинарными или двойными кавычками, чтобы они передавались одной группой.

`-o` *параметры-initdb*
`--options=`*параметры-initdb*

Указывает параметры, которые будут передаваться непосредственно программе `initdb`. Ключ `-o` можно указывать несколько раз, при этом ей будут переданы параметры из всех ключей.

Задаваемые *параметры-initdb* обычно следует обрамлять одинарными или двойными кавычками, чтобы они передавались вместе одной группой.

`-p` *путь*

Указывает размещение исполняемого файла `postgres`. По умолчанию задействуется исполняемый файл `postgres` из того же каталога, из которого запускался `pg_ctl`, а если это невозможно, из жёстко заданного каталога инсталляции. Применять этот параметр может понадобиться, только если вы делаете что-то необычное или получаете сообщения, что найти исполняемый файл `postgres` не удаётся.

В режиме `init` этот параметр аналогичным образом задаёт размещение исполняемого файла `initdb`.

`-s`
`--silent`

Выводить лишь ошибки, без сообщений информационного характера.

`-t` *секунды*
`--timeout=`*секунды*

Задаёт максимальное время (в секундах) ожидания завершения операции (см. параметр `-w`). По умолчанию действует значение переменной среды `PGCTLTIMEOUT` или, если оно не задано, 60 секунд.

`-V`
`--version`

Выводит версию pg_ctl и прерывает выполнение.

`-w`
`--wait`

Ждать завершения операции. Этот режим поддерживается (и действует по умолчанию) для команд `start`, `stop`, `restart`, `promote` и `register`.

В процессе ожидания `pg_ctl` постоянно проверяет PID-файл сервера, приостанавливаясь на короткое время между проверками. Запуск считается завершённым, когда PID-файл указывает на то, что сервер готов принимать подключения. Остановка считается завершённой, когда

сервер удаляет свой PID-файл. Программа `pg_ctl` возвращает код завершения в зависимости от успеха запуска или остановки.

Если операция не заканчивается за отведённое время (см. параметр `-t`), программа `pg_ctl` завершается с ненулевым кодом выхода. Но заметьте, что при этом выполнение операции может продолжиться и в конце концов увенчаться успехом.

`-W`
`--no-wait`

Не ждать завершения операции. Этот режим противоположен режиму `-w`.

Если ожидание отключено, запрошенное действие вызывается, но о его результате ничего не известно. В этом случае для проверки текущего состояния и результата операции потребуется обратиться к файлу журнала сервера или воспользоваться внешней системой мониторинга.

В предыдущих выпусках Postgres Pro этот режим действовал по умолчанию (кроме команды `stop`).

`-?`
`--help`

Вывести справку по команде pg_ctl и прервать выполнение.

Если некоторый параметр является допустимым, но не применим к выбранному режиму работы, pg_ctl игнорирует его.

Параметры, специфичные для Windows

`-e` *`source`*

Имя источника событий, с которым pg_ctl будет записывать в системный журнал события при запуске в виде службы Windows. Имя по умолчанию — `Postgres Pro`. Заметьте, что это влияет только на сообщения, которые выдаёт сам pg_ctl; как только сервер запустится, он будет использовать источник событий, заданный в event_source. Если произойдёт ошибка при запуске сервера на ранней стадии, прежде чем будет считан этот параметр, он может также выдавать сообщения с источником по умолчанию `Postgres Pro`.

`-N` *`имя_службы`*

Имя регистрируемой системной службы. Оно станет и собственно именем службы, и отображаемым именем. По умолчанию — `Postgres Pro`.

`-P` *`пароль`*

Пароль для пользователя, запускающего службу.

`-S` *`тип-запуска`*

Тип запуска системной службы. В качестве значения *`тип-запуска`* можно задать `auto`, `demand` или первую букву этих слов. По умолчанию выбирается тип `auto`.

`-U` *`имя_пользователя`*

Имя пользователя, от имени которого будут запущена служба. Для доменных пользователей используйте формат `DOMAIN\username`.

Переменные окружения

`PGCTLTIMEOUT`

Значение по умолчанию для максимального времени ожидания запуска или остановки сервера (в секундах). По умолчанию это время составляет 60 секунд.

PGDATA

Размещение каталога хранения данных по умолчанию.

Для большинства режимов pg_ctl требуется знать расположение каталога данных; поэтому если не задана переменная PGDATA, параметр -D является обязательным.

pg_ctl, как и большинство других утилит Postgres Pro, также использует переменные окружения, поддерживаемые libpq (см. Раздел 31.14).

Список дополнительных переменных, влияющих на работу сервера, можно найти в postgres.

Файлы

postmaster.pid

Проверяя этот файл в каталоге данных, pg_ctl определяет, работает ли сервер в настоящий момент.

postmaster.opts

Если файл существует в каталоге хранения данных, то pg_ctl (при restart) передаст его содержимое в качестве аргументов postgres, если не указаны иные значения в -o. Содержимое файла также отображается при вызове в режиме status.

Примеры

Запуск сервера

Запуск сервера и ожидание момента, когда он начнёт принимать подключения:

```
$ pg_ctl start
```

Чтобы запустить сервер с использованием порта 5433 и без fsync, выполните:

```
$ pg_ctl -o "-F -p 5433" start
```

Остановка сервера

Чтобы остановить сервер, выполните:

```
$ pg_ctl stop
```

Ключ -m позволяет управлять тем, *как* сервер будет остановлен:

```
$ pg_ctl stop -m smart
```

Повторный запуск сервера

Перезапуск сервера почти равнозначен остановке и запуску сервера за исключением того, что по умолчанию pg_ctl сохраняет параметры командной строки, которые были переданы ранее запущенному экземпляру. Таким образом, чтобы перезапустить сервер с теми же параметрами, с какими он был запущен, выполните:

```
$ pg_ctl restart
```

Но если добавляется ключ -o, он заменяет все предыдущие параметры. Эта команда осуществит перезапуск с использованием порта 5433 и без fsync:

```
$ pg_ctl -o "-F -p 5433" restart
```

Вывод состояния сервера

Ниже представлен примерный вывод pg_ctl:

```
$ pg_ctl status
```

```
pg_ctl: server is running (PID: 13718)
/usr/local/pgsql/bin/postgres "-D" "/usr/local/pgsql/data" "-p" "5433" "-B" "128"
```

Во второй строке показывается команда, которая будет выполнена в режиме перезапуска.

См. также

initdb, postgres

pg_resetwal

pg_resetwal — очистка журнала предзаписи и другой управляющей информации кластера Postgres Pro

Синтаксис

`pg_resetwal` [`-f`] [`-n`] [*параметр*...] {[`-D`] *каталог_данных*}

Описание

`pg_resetwal` очищает журнал предзаписи (WAL) и может сбросить некоторую другую управляющую информацию, хранящуюся в файле `pg_control`. Данная функция может быть востребована при повреждении этих файлов. Использовать её нужно только как крайнюю меру, когда запуск сервера оказывается невозможен из-за этого повреждения.

После выполнения этой команды запуск сервера, скорее всего, будет возможен, однако стоит учитывать, что база данных может содержать несогласованные данные из-за транзакций, зафиксированных частично. Вы должны немедленно выгрузить данные, выполнить `initdb`, а затем восстановить данные. После этого проверьте целостность базы и внесите необходимые коррективы.

Эту утилиту может запускать только пользователь, установивший сервер, так как ей нужны права записи/чтения в каталоге хранения данных кластера. В целях безопасности каталог необходимо указывать в командной строке. `pg_resetwal` не поддерживает переменную окружения `PGDATA`.

Если `pg_resetwal` сообщает о невозможности определить данные из `pg_control`, команду можно запустить принудительно, указав `-f`. В этом случае будут использованы наиболее вероятные значения. Они должны подходить для большинства полей, но для некоторых может потребоваться задать нужные значения явно: следующее значение OID, ID и эпоха следующей транзакции, ID мультитранзакции и смещение, начальный адрес WAL. Эти значения можно указать с помощью описанных далее параметров. Если их невозможно определить, то флаг `f` позволяет это обойти. Однако достоверность данных восстановленной базы не гарантируется: крайне необходимо незамедлительно выгрузить и затем восстановить данные. *Не* выполняйте никаких операций модификации до создания дампа данных, так как это может привести к ещё более печальным последствиям.

Параметры

`-f`

Принудительно выполнять `pg_resetwal`, даже если не удаётся получить приемлемые данные из `pg_control`, как описано выше.

`-n`

С флагом `-n` (нет операции) команда `pg_resetwal` отображает извлечённые из `pg_control` данные, а также значения, которые можно изменить. Режим полезен для отладки и тестирования предстоящей операции без реального применения изменений.

`-V`
`--version`

Показать версию, а затем завершиться.

`-?`
`--help`

Показать справку, а затем завершиться.

Следующие параметры необходимы, только когда pg_resetwal не может определить подходящие значения, прочитав pg_control. Безопасные значения можно определить, как описано ниже. Для значений, принимающих числовые аргументы, можно задать шестнадцатеричные значения, добавив префикс 0x.

-c *xid*,*xid*

Вручную задать идентификаторы старейшей и новейшей транзакций, для которых можно получить время фиксации.

Безопасное значение идентификатора старейшей транзакции, для которой можно получить время фиксации (первый компонент), можно определить, найдя наименьшее в числовом виде имя файла в каталоге pg_commit_ts внутри каталога данных. Безопасное значение идентификатора новейшей транзакции, для которой можно получить время фиксации (второй компонент), можно определить, найдя, напротив, наибольшее в числовом виде имя файла в том же каталоге. Числа в именах этих файлов представлены в шестнадцатеричном формате.

-e *эпоха_xid*

Вручную задать эпоху в ID следующей транзакции.

Эпоха идентификаторов транзакции не хранится в базе данных нигде, кроме поля, устанавливаемого командой pg_resetwal, поэтому если рассматривать собственно базу, допустимым будет любое значение. Это значение, возможно, понадобится скорректировать для обеспечения правильной работы системы репликации, например, Slony-I и Skytools. В этом случае подходящее значение следует получить из состояния нижележащей реплицированной базы данных.

-l *walfile*

Вручную задать начальный адрес WAL.

Начальный адрес журнала WAL должен превышать наибольшее значение сегмента в имени файла, расположенного в каталоге pg_wal. Эти имена тоже представлены в шестнадцатеричном формате и состоят из трёх частей. Первая из них — «ID линии времени», и её обычно не следует менять. Например, если 000000010000000320000004A — наибольшее значение в pg_wal, нужно указать -l 000000010000000320000004B или большее число.

> **Примечание**
>
> pg_resetwal ищет среди файлов каталога pg_wal, и по умолчанию выбирает значение для флага -l, идущее следующим после найденного. Таким образом, вручную задавать параметр -l нужно лишь если известно о существовании сегментов WAL, отсутствующих в настоящий момент в каталоге pg_wal (например, они могут находится в отдельном архиве); либо если содержимое pg_wal было полностью утеряно.

-m *mxid*,*mxid*

Вручную задать ID следующей и старейшей мультитранзакции.

Безопасное значение следующего идентификатора мультитранзакции (первый компонент) можно вычислить, найдя наибольшее числовое значение среди имён файлов, расположенных в каталоге pg_multixact/offsets. К найденному значению необходимо прибавить один, затем умножить на 65536 (0x10000). Для вычисления безопасного значения ID старейшей мультитранзакции (второй компонент -m) необходимо найти наименьшее числовое значение среди тех же файлов, и умножить его на 65536. Имена файлов представляются в шестнадцатеричном формате, так что значения проще указывать в нём же, добавив в конце четыре нуля.

`-o` *oid*

Вручную задать следующий OID.

Не существует относительно простого способа вычисления следующего за наибольшим из существующих значением OID, однако это некритично.

`-O` *mxoff*

Вручную задать смещение следующей мультитранзакции.

Безопасное значение можно определить, найдя наибольшее числовое значение в имени файла в каталоге `pg_multixact/members`, прибавив один и умножив результат на 52352 (0xCC80). Имена файлов представлены в шестнадцатеричном формате. Простого рецепта с прибавлением нулей в конце, как для других параметров, в данном случае нет.

`-u` *xid*

Вручную задать ID старейшей незамороженной транзакции.

Безопасное значение можно определить, найдя наименьшее числовое значение в имени файла в подкаталоге `pg_xact` каталога данных, а затем умножив результат на 1048576 (0x100000). Заметьте, что имена файлов представлены в шестнадцатеричном формате. Значение этого параметра обычно также проще задавать в шестнадцатеричном виде. Например, если `0007` — наименьшее значение в `pg_xact`, подходящим значением будет `-u 0x700000` (нужный множитель дают пять последних нулей).

`-x` *xid*

Вручную задать ID следующей транзакции.

Безопасное значение можно определить, найдя наибольшее числовое значение в имени файла в подкаталоге `pg_xact` каталога данных, прибавив один и умножив результат на 1048576 (0x100000). Заметьте, что имена файлов представлены в шестнадцатеричном формате. Значение этого параметра обычно также проще задавать в шестнадцатеричном виде. Например, если `0011` — наибольшее значение в `pg_xact`, подходящим значением будет `-x 0x1200000` (нужный множитель дают пять последних нулей).

Замечания

Эту команду нельзя выполнять на работающем сервере. `pg_resetwal` отклонит выполнение при обнаруженном блокирующем файле в каталоге хранения данных. Иногда при аварии сервера блокирующий файл может остаться в системе. В этом случае необходимо самостоятельно удалить его, чтобы дать возможность `pg_resetwal` отработать. Перед выполнением операции дважды проверьте, что сервер остановлен.

`pg_resetwal` работает только с серверами той же основной версии.

См. также

pg_controldata

pg_rewind

pg_rewind — синхронизировать каталог данных Postgres Pro с другим каталогом, ответвлённым от него

Синтаксис

`pg_rewind` [*параметр*...] { `-D` | `--target-pgdata` }*каталог* { `--source-pgdata=`*каталог* | `--source-server=`*строка_подключения* }

Описание

Утилита pg_rewind представляет собой средство синхронизации кластера Postgres Pro с другой копией того же кластера после расхождения линий времени этих кластеров. Обычный сценарий её использования — вернуть в работу старый главный сервер после переключения на резервный, в качестве резервного для сервера, ставшего главным.

Её результат равнозначен замене целевого каталога данных исходным. В файлах отношений она копирует только изменённые блоки; все остальные файлы, включая файлы конфигурации, копируются целиком. Преимущество pg_rewind по сравнению с созданием новой базовой копии или такими средствами, как rsync, состоит в том, что pg_rewind не читает неизменённые блоки в кластере. Благодаря этому она действует гораздо быстрее, когда база данных большая, но различия между кластерами ограничиваются лишь небольшим количеством блоков.

Утилита pg_rewind изучает истории линий времени исходного и целевого кластеров с целью найти точку, в которой они разошлись, и ожидает найти журналы WAL в каталоге `pg_wal` целевого кластера вплоть до точки расхождения. Точка расхождения может быть найдена на целевой или исходной линии времени либо в их общем предке. В типичном сценарии отработки отказа, когда целевой кластер отключается вскоре после расхождения, это не проблема, но если целевой кластер проработал долгое время после расхождения, старые файлы WAL могут быть уже удалены. В этом случае их можно вручную скопировать из архива WAL в каталог `pg_wal`. Варианты использования pg_rewind не ограничиваются отработкой отказа; например, резервный сервер может быть повышен, выполнить несколько транзакций с записью, а затем, после восстановления синхронизации, вновь стать резервным.

Когда целевой сервер запускается в первый раз после выполнения pg_rewind, он переходит в режим восстановления и воспроизводит все изменения из WAL с исходного сервера после точки расхождения. Если какие-то сегменты WAL оказались недоступны на исходном сервере, когда выполнялась pg_rewind, и поэтому не могли быть скопированы в ходе работы pg_rewind, их необходимо предоставить, когда сервер будет запускаться. Это можно сделать, создав в целевом каталоге данных файл `recovery.conf` с подходящей командой `restore_command`.

Утилита pg_rewind требует, чтобы на целевом сервере был либо включён режим wal_log_hints в `postgresql.conf`, либо включены контрольные суммы, когда кластер был инициализирован командой initdb. Оба эти режима по умолчанию отключены. Также должен быть включён режим full_page_writes, но он по умолчанию включён.

Предупреждение

Если во время работы pg_rewind происходит ошибка, вероятнее всего, целевой каталог данных будет в состоянии, не подходящем для восстановления. В этом случае рекомендуется сделать новую резервную копию.

Программа pg_rewind немедленно прекращает работу, если обнаруживает файлы, непосредственная запись в которые невозможна. Это может иметь место, например, когда на исходном и целевом серверах совпадают пути файлов сертификатов и ключей SSL, доступных только для чтения. Если такие файлы существуют на целевом сервере, их рекомендуется

удалить до запуска pg_rewind. После выполнения синхронизации некоторые из таких файлов могут быть скопированы из источника и тогда может потребоваться удалить скопированные данные и восстановить ссылки/файлы, существовавшие до синхронизации.

Параметры

pg_rewind принимает следующие аргументы командной строки:

`-D` *`каталог`*
`--target-pgdata=`*`каталог`*

Этот параметр задаёт целевой каталог данных, который будет синхронизирован с источником. Целевой сервер должен быть отключён штатным образом до запуска pg_rewind

`--source-pgdata=`*`каталог`*

Задаёт путь в файловой системе к каталогу данных исходного сервера, с которым будет синхронизироваться целевой. Для применения этого ключа исходный сервер должен быть остановлен штатным образом.

`--source-server=`*`строка_подключения`*

Задаёт строку подключения libpq для подключения к исходному серверу Postgres Pro, с которым будет синхронизирован целевой. Подключение должно устанавливаться как обычное (не реплицирующее) с правами суперпользователя. Для применения этого параметра исходный сервер должен быть запущен и работать не в режиме восстановления.

`-n`
`--dry-run`

Делать всё, кроме внесения изменений в целевой каталог.

`-P`
`--progress`

Включает вывод сообщений о прогрессе. При этом в процессе копирования данных из исходного кластера будет выдаваться приблизительный процент выполнения.

`--debug`

Выводить подробные отладочные сообщения, полезные в основном для разработчиков, отлаживающих pg_rewind.

`-V`
`--version`

Показать версию, а затем завершиться.

`-?`
`--help`

Показать справку, а затем завершиться.

Переменные окружения

Когда используется `--source-server`, pg_rewind также использует переменные среды, поддерживаемые libpq (см. Раздел 31.14).

Замечания

Когда исходным кластером для pg_rewind является работающий сервер сразу после повышения, в нём необходимо выполнить команду `CHECKPOINT`, чтобы его управляющий файл содержал

актуальную информацию о линии времени. Эта информацию нужна pg_rewind для проверки, можно ли синхронизировать целевой кластер с выбранным исходным.

Как это работает

Основная идея состоит в том, чтобы перенести все изменения на уровне файловой системы из исходного кластера в целевой:

1. Просканировать журнал WAL в целевом кластере, начиная с последней контрольной точки перед моментом, когда история линии времени исходного кластера разошлась с целевым кластером. Для каждой записи WAL отметить, какие блоки данных были затронуты. В результате будет получен список всех блоков данных, которые были изменены в целевом кластере после отделения исходного.
2. Скопировать все эти изменённые блоки из исходного кластера в целевой либо на уровне файловой системы (`--source-pgdata`), либо на уровне SQL (`--source-server`).
3. Скопировать все остальные файлы, в частности `pg_xact` и файлы конфигурации, из исходного кластера в целевой (пропуская при этом файлы отношений).
4. Применить WAL из исходного кластера, начиная с контрольной точки, созданной при переключении. (Строго говоря, утилита pg_rewind не применяет WAL, она просто создаёт файл метки резервной копии, обнаружив который, Postgres Pro начинает воспроизведение всех записей WAL от этой контрольной точки.)

pg-setup

pg-setup — создать кластер баз данных Postgres Pro и управлять соответствующей службой

Синтаксис

`pg-setup initdb [`*`параметры_initdb`*`]`

`pg-setup find-free-port`

`pg-setup set-server-port` *`порт`*

`pg-setup set` *`параметр значение`*

`pg-setup service` *`параметр_service`*

`pg-setup tune` *`конфигурация`*

Описание

Программа pg-setup представляет собой скрипт оболочки, включённый в дистрибутив Postgres Pro и предназначенный для автоматизации установки кластера баз данных в системах Linux. Этот скрипт входит в состав пакета `postgrespro-std-10-server`. После установки Postgres Pro вы можете найти pg-setup в каталоге *`каталог-инсталляции`*`/bin`, где *`каталог-инсталляции`* обозначает путь `/opt/pgpro/std-10`.

Программу pg-setup нужно запускать от имени root, но она выполняет административные операции с базой данных под именем пользователя `postgres`. Запуская её с разными параметрами, вы можете:

- инициализировать кластер баз данных
- проверить доступные порты и изменить порт, который будет использоваться сервером Postgres Pro
- изменить параметры конфигурации сервера
- включить/отключить автоматический запуск службы Postgres Pro
- запустить, остановить или перезапустить службу Postgres Pro
- настроить кластер БД для определённого продукта Postgres Pro

Параметры

pg-setup принимает следующие аргументы командной строки:

`initdb [`*`параметры_initdb`*`]`

Инициализировать кластер баз данных от имени пользователя `postgres`.

По умолчанию кластер баз данных инициализируется в каталоге `/var/lib/pgpro/std-10/data` и настраивается для продукта Postgres Pro; в этом кластере будут включены контрольные суммы, параметр `auth-local` будет иметь значение `peer`, а `auth-host` — значение `md5`. Параметры локализации наследуются из переменной окружения `LANG`, определённой в текущем сеансе. Все переменные окружения `LC_*` при этом игнорируются. Вы можете также передать параметры initdb для дополнительной настройки инсталляции.

Если база данных по умолчанию создаётся с помощью pg-setup, путь к каталогу данных сохраняется в файле `/etc/default/postgrespro-standard-10`, чтобы все последующие команды pg-setup, а также все команды, управляющие службой Postgres Pro, воздействовали именно на эту базу данных. Управлять несколькими базами данных с помощью pg-setup нельзя.

`find-free-port`

Найти свободный порт в системе. Это может быть полезно, если вы планируете устанавливать несколько экземпляров серверов, либо стандартный порт 5432 уже используется другой программой.

`set-server-port` *порт*

Задать номер порта, на котором сервер будет принимать соединения. Воспользуйтесь этой командой, чтобы избежать конфликтов, если вы устанавливаете в одной системе более одного сервера.

По умолчанию: 5432

`set` *имя значение*

Установить для заданного параметра конфигурации указанное *значение* в файле `postgresql.conf`. Если этот параметр ранее был определён командой `ALTER SYSTEM`, его предыдущее значение удаляется из файла `postgresql.auto.conf`.

`service` *параметр_service*

Управление службой Postgres Pro с помощью следующих команд:

- `enable` — включить автоматический запуск службы при перезагрузке системы.
- `disable` — отключить автоматический запуск службы при перезагрузке системы.
- `start` — запустить службу.
- `stop` — остановить службу.
- `condrestart` — перезапустить службу, если она работает в момент вызова pg-setup.
- `status` — возвращает статус службы Postgres Pro.

`tune` *конфигурация*

Выбирает заданную конфигурацию для кластера баз данных. Предопределены следующие значения *конфигурации* (их доступность зависит от редакции Postgres Pro): `1c`, `std` или `ent`. Значение `1c` допускается для любого продукта. По умолчанию выбирается конфигурация, определяемая устанавливаемой редакцией Postgres Pro Standard. Если вы укажете какое-либо другое, не предопределённое, значение, кластер будет использовать стандартную конфигурацию.

В случае, если в вашей системе будут работать несколько серверов БД и/или приложений, может потребоваться изменить настроенную конфигурацию, так как pg_setup выбирает параметры конфигурации в зависимости от характеристик оборудования, полагая, что система будет использоваться только одним сервером БД.

Замечания

Если вы устанавливаете Postgres Pro из пакета `postgrespro-std-10`, pg-setup вызывается автоматически с параметрами по умолчанию. В результате создаётся кластер баз данных и база по умолчанию в каталоге `/var/lib/pgpro/std-10/data`, настраивается автозапуск службы Postgres Pro и служба запускается.

Если вы устанавливаете сервер Postgres Pro непосредственно из пакета `postgrespro-std-10-server`, вы можете запустить этот скрипт вручную для инициализации кластера баз данных или управления службой Postgres Pro.

Установка из двоичных пакетов в Linux подробно описана в Разделе 16.1.

pg_test_fsync

pg_test_fsync — подобрать наилучший вариант `wal_sync_method` для Postgres Pro

Синтаксис

`pg_test_fsync` [*параметр*...]

Описание

Программа pg_test_fsync предназначена для того, чтобы дать вам представление о том, какой из вариантов wal_sync_method оптимален для вашей конкретной системы, а также выдать вспомогательные диагностические сведения в случае проблем со вводом/выводом. Однако отличия, показанные программой pg_test_fsync, могут не оказывать большого влияния на реальную производительность баз данных, в частности потому, что для многих серверов баз данных производительность упирается не в запись журналов предзаписи. pg_test_fsync выводит среднее время операции синхронизации с ФС для каждого метода `wal_sync_method`, что также может быть полезно при поиске оптимального значения commit_delay.

Параметры

pg_test_fsync принимает следующие параметры командной строки:

`-f`
`--filename`

Задаёт имя файла для записи данных тестов. Этот файл должен находиться в той же файловой системе, где размещается или будет размещаться каталог `pg_wal`. (В каталоге `pg_wal` содержатся файлы WAL.) По умолчанию выбирается файл `pg_test_fsync.out` в текущем каталоге.

`-s`
`--secs-per-test`

Задаёт продолжительность каждого теста (в секундах). Чем больше длится тест, тем точнее результат, но тем дольше работает программа. Со значением по умолчанию (5 секунд) программа должна завершиться примерно за 2 минуты.

`-V`
`--version`

Вывести версию pg_test_fsync и завершиться.

`-?`
`--help`

Вывести справку об аргументах командной строки pg_test_fsync и завершиться.

См. также

postgres

pg_test_timing

pg_test_timing — определить издержки замера времени

Синтаксис

```
pg_test_timing [параметр...]
```

Описание

Программа pg_test_timing позволяет оценить издержки замера времени в вашей системе и убедиться в том, что системное время никогда не идёт назад. Системы, в которых замер времени является длительной операцией, дают менее точные результаты `EXPLAIN ANALYZE`.

Параметры

pg_test_timing принимает следующие аргументы командной строки:

`-d` *длительность*
`--duration=`*длительность*

Задаёт продолжительность теста (в секундах). Чем больше эта продолжительность, тем выше точность и больше вероятность обнаружить аномалию с обратным ходом системных часов. По умолчанию время тестирования — 3 секунды.

`-V`
`--version`

Вывести версию pg_test_timing и завершиться.

`-?`
`--help`

Вывести справку об аргументах командной строки pg_test_timing и завершиться.

Использование

Интерпретация результатов

В благоприятном случае практически все (>90%) отдельные вызовы замеров времени должны выполняться быстрее одной микросекунды. Средние издержки замера на цикл должны быть ещё меньше, в пределах 100 наносекунд. Эта проба, взятая в системе Intel i7-860 через источник времени TSC, показывает отличную производительность:

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 35.96 ns
Histogram of timing durations:
  < us   % of total      count
     1     96.40465   80435604
     2      3.59518    2999652
     4      0.00015        126
     8      0.00002         13
    16      0.00000          2
```

Заметьте, что время вызова в цикле и время в гистограмме выражается в разных единицах. Время в цикле может определяться с точностью до наносекунд (ns), а длительность отдельного вызова замера времени — только с точностью до микросекунд (us).

Измерение издержек исполнителя на замер времени

Когда исполнитель запроса выполняет запрос под контролем `EXPLAIN ANALYZE`, замеряется не только общее время, но и время отдельных операций. Каковы издержки этих операций в вашей системе, можно узнать, подсчитав строки тестовой таблицы в программе psql:

```
CREATE TABLE t AS SELECT * FROM generate_series(1,100000);
\timing
SELECT COUNT(*) FROM t;
EXPLAIN ANALYZE SELECT COUNT(*) FROM t;
```

В системе i7-860 этот запрос выполняется 9.8 мс, а версия с `EXPLAIN ANALYZE` — 16.6 мс, при этом обрабатывается около 100 000 строк. Это различие в 6.8 мс означает, что издержки замера времени для одной строки составляют около 68 нс, примерно вдвое больше, чем предсказала pg_test_timing. Даже с такими относительно небольшими издержками операция COUNT с полным подсчётом времени выполняется почти на 70% дольше. На более сложных запросах издержки замера времени могут быть не так важны.

Смена источника времени

В некоторых современных системах Linux можно в любой момент сменить источник времени, который используется для замера времени. Второй пример иллюстрирует возможное замедление от переключения на более медленный источник времени acpi_pm time в той же системе, в которой были получены показанные выше хорошие результаты:

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
# echo acpi_pm > /sys/devices/system/clocksource/clocksource0/current_clocksource
# pg_test_timing
Per loop time including overhead: 722.92 ns
Histogram of timing durations:
  < us   % of total      count
     1     27.84870    1155682
     2     72.05956    2990371
     4      0.07810       3241
     8      0.01357        563
    16      0.00007          3
```

В этой конфигурации тот же `EXPLAIN ANALYZE` выполняется 115.9 мс. Таким образом издержки составили 1061 нс, что соответствует непосредственному результату этой утилиты с небольшим коэффициентом. Такие большие издержки означают, что сам запрос выполняется лишь небольшой процент всего времени, а основное время уходит на замеры времени. В такой конфигурации временные показатели `EXPLAIN ANALYZE` для запросов со множеством замеряемых операций значительно увеличатся за счёт издержек замера времени.

FreeBSD так же позволяет сменять источник времени «на лету» и выводит информацию о выбранном таймере при загрузке:

```
# dmesg | grep "Timecounter"
Timecounter "ACPI-fast" frequency 3579545 Hz quality 900
Timecounter "i8254" frequency 1193182 Hz quality 0
Timecounters tick every 10.000 msec
Timecounter "TSC" frequency 2531787134 Hz quality 800
# sysctl kern.timecounter.hardware=TSC
kern.timecounter.hardware: ACPI-fast -> TSC
```

Другие системы могут допускать смену источника времени только при загрузке. В старых системах Linux это можно было сделать только с помощью параметра ядра «clock». И даже в некоторых самых последних системах можно увидеть только один источник времени — jiffies. Это старая программная реализация часов в Linux, которая может давать хорошее разрешение, когда поддерживается достаточно хорошим оборудованием, как в этом примере:

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
jiffies
$ dmesg | grep time.c
time.c: Using 3.579545 MHz WALL PM GTOD PIT/TSC timer.
time.c: Detected 2400.153 MHz processor.
```

```
$ pg_test_timing
Testing timing overhead for 3 seconds.
Per timing duration including loop overhead: 97.75 ns
Histogram of timing durations:
  < us   % of total      count
     1     90.23734   27694571
     2      9.75277    2993204
     4      0.00981       3010
     8      0.00007         22
    16      0.00000          1
    32      0.00000          1
```

Аппаратные часы и точность замера времени

Измерение времени обычно осуществляется на компьютерах по аппаратным часам, точность которых может быть разного уровня. С некоторым оборудованием операционные системы могут передавать время системных часов непосредственно программам. Также системное время может поступать с чипа, который просто генерирует прерывания по времени, с заведомо известным периодом. В любом случае ядра операционных систем предоставляют источник времени, который скрывает эти детали. Но точность этого источника и возможная скорость получения результатов от него зависит от нижележащего оборудования.

Неточность в замерах времени может приводить к нестабильности системы. Поэтому стоит очень тщательно протестировать выбранный источник времени. Иногда по умолчанию в ОС выбирается источник не более точный, а более надёжный. И если вы используете виртуальную машину, поинтересуйтесь, какие источники времени рекомендуется использовать с ней. Имитация таймеров на виртуальном оборудовании связана с дополнительными сложностями, и производители средств виртуализации часто рекомендуют определённые параметры для операционных систем.

Источник времени TSC (Time Stamp Counter, Счётчик отметки времени) наиболее точный из всех для процессоров текущего поколения. Его рекомендуется использовать для получения системного времени, когда он поддерживается операционной системой и показания TSC надёжны. Возможны ситуации, когда TSC не является точным источником времени, и таким образом, оказывается ненадёжным. Например, в старых системах показания TSC могут зависеть от температуры процессора, что не годится для точного замера времени. При попытке использовать TSC в некоторых старых многоядерных процессорах можно получить разное время на различных ядрах. В результате может оказаться, что время идёт назад (эту аномалию выявляет данная программа). И даже на самых современных системах не всегда можно получить точное время через TSC в режимах очень агрессивного энергосбережения.

Новые операционные системы могут проверять наличие известных проблем TSC и переключаться на более медленный, но более стабильный источник времени, если они проявляются. Если ваша система поддерживает источник TSC, но не выбирает его по умолчанию, возможно, он отключён обоснованно. С другой стороны, некоторые операционные системы могут не выявлять все возможные проблемы, либо разрешают использовать TSC даже в ситуациях, когда он определённо неточен.

HPET (High Precision Event Timer, Таймер событий высокой точности) рекомендуется использовать в системах, где он имеется, а TSC неточен. Сам этот чип можно запрограммировать для получения точности до 100 наносекунд, но системное время с такой точностью вы не получите.

ACPI (Advanced Configuration and Power Interface, Расширенный интерфейс конфигурации и питания) обеспечивает таймер PM (Управления питанием), который в Linux называется acpi_pm. Время, поступающее из acpi_pm, в лучшем случае будет иметь разрешение 300 наносекунд.

На старом оборудовании PC применялись таймеры 8254 PIT (Programmable Interval Timer, Программируемый интервальный таймер), RTC (Real-Time Clock, Часы реального времени), таймер APIC (Advanced Programmable Interrupt Controller, Расширенный программируемый контроллер прерываний) и Cyclone. Все эти таймеры обеспечивали точность до миллисекунд.

См. также

EXPLAIN

pg_upgrade

pg_upgrade — обновить экземпляр сервера Postgres Pro

Синтаксис

pg_upgrade -b *старый_каталог_bin* -B *новый_каталог_bin* -d *старый_каталог_конфигурации* -D *новый_каталог_конфигурации* [*параметр*...]

Описание

Программа pg_upgrade (ранее называвшаяся pg_migrator) позволяет обновить данные в каталоге базы PostgreSQL или Postgres Pro до последней основной версии Postgres Pro без операции выгрузки/перезагрузки данных, обычно необходимой при обновлениях основной версии, например, при переходе от 9.5.8 к 9.6.4 или от 10.7 к 11.2. Эти действия не требуются при установке корректирующей версии, например, при переходе от 9.6.2 к 9.6.3 или от 10.1 к 10.2.

С выходом новых основных версий в Postgres Pro регулярно добавляются новые возможности, которые часто меняют структуру системных таблицы, но внутренний формат хранения меняется редко. Учитывая этот факт, pg_upgrade позволяет выполнить быстрое обновление, создавая системные таблицы заново, но сохраняя старые файлы данных. Если при обновлении основной версии формат хранения данных изменится так, что данные в старом формате окажутся нечитаемыми, pg_upgrade не сможет произвести такое обновление. (Сообщество разработчиков постарается не допустить подобных ситуаций.)

Программа pg_upgrade делает всё возможное, чтобы убедиться в том, что старый и новый кластеры двоично-совместимы, в частности проверяя параметры времени компиляции и разрядность (32/64 бита) исполняемых файлов. Важно, чтобы и все внешние модули тоже были двоично-совместимыми, хотя это pg_upgrade проверить не может.

pg_upgrade поддерживает обновление с версии 8.4.X и новее до текущей основной версии Postgres Pro, включая бета-выпуски и сборки снимков кода.

Параметры

pg_upgrade принимает следующие аргументы командной строки:

-b *каталог_bin*
--old-bindir=*каталог_bin*

каталог с исполняемыми файлами старой версии Postgres Pro; переменная окружения PGBINOLD

-B *каталог_bin*
--new-bindir=*каталог_bin*

каталог с исполняемыми файлами новой версии Postgres Pro; переменная окружения PGBINNEW

-c
--check

только проверить кластеры, не изменять никакие данные

-d *каталог_конфигурации*
--old-datadir=*каталог_конфигурации*

каталог конфигурации старого кластера; переменная окружения PGDATAOLD

-D *каталог_конфигурации*
--new-datadir=*каталог_конфигурации*у

каталог конфигурации нового кластера; переменная окружения PGDATANEW

`-j` *число_заданий*
`--jobs=`*число_заданий*

число одновременно задействуемых процессов или потоков

`-k`
`--link`

использовать жёсткие ссылки вместо копирования файлов в новый кластер

`-o` *параметры*
`--old-options` *параметры*

параметры, передаваемые непосредственно старой программе `postgres`; несколько параметров складываются вместе

`-O` *параметры*
`--new-options` *параметры*

параметры, передаваемые непосредственно новой программе `postgres`; несколько параметров складываются вместе

`-p` *порт*
`--old-port=`*порт*

номер порта старого кластера; переменная окружения `PGPORTOLD`

`-P` *порт*
`--new-port=`*порт*

номер порта нового кластера; переменная окружения `PGPORTNEW`

`-r`
`--retain`

сохранить SQL и журналы сообщений даже при успешном завершении

`-U` *имя_пользователя*
`--username=`*имя_пользователя*

имя пользователя, установившего кластер; переменная окружения `PGUSER`

`-v`
`--verbose`

включить подробные внутренние сообщения

`-V`
`--version`

показать версию, а затем завершиться

`-?`
`--help`

показать справку, а затем завершиться

Использование

Далее описан план обновления с использованием pg_upgrade:

1. **Переместить старый кластер (необязательно)**

   Если ваш каталог инсталляции привязан к версии, например, `/opt/PostgreSQL/10`, перемещать его не требуется. Все графические инсталляторы выбирают при установке каталоги, привязанные к версии.

Если ваш каталог инсталляции не привязан к версии, например `/usr/local/pgsql`, необходимо переместить каталог текущей инсталляции Postgres Pro, чтобы он не конфликтовал с новой инсталляцией Postgres Pro. Когда текущий сервер Postgres Pro отключён, каталог этой инсталляции Postgres Pro можно безопасно переместить; если старый каталог `/usr/local/pgsql`, его можно переименовать, выполнив:

```
mv /usr/local/pgsql /usr/local/pgsql.old
```

2. **Установить новые исполняемые файлы Postgres Pro**

   Установите новые исполняемые файлы сервера и вспомогательные файлы. Программа pg_upgrade включена в инсталляцию по умолчанию.

3. **Инициализировать новый кластер Postgres Pro**

   Инициализируйте новый кластер, используя `initdb`. При этом так же необходимо указать флаги `initdb`, совместимые с флагами в старом кластере. Многие готовые инсталляторы выполняют это действие автоматически. Запускать новый кластер не требуется.

4. **Установить разделяемые объектные файлы расширения**

   Многие расширения и пользовательские модули, как из `contrib`, так и из других источников, используют разделяемые объектные файлы (или библиотеки DLL), например, `pgcrypto.so`. Если они использовались в старом кластере, разделяемые объектные файлы, соответствующие новому исполняемому файлу сервера, должны быть установлены в новом кластере, обычно средствами операционной системы. Не загружайте определения схемы, например, выполняя `CREATE EXTENSION pgcrypto`, потому что они будут скопированы из старого кластера. Если доступны обновления расширений, pg_upgrade сообщит об этом и создаст скрипт, который можно будет запустить позже, чтобы обновить эти расширения.

5. **Скопируйте пользовательские файлы полнотекстового поиска**

   Скопировать пользовательские файлы полнотекстового поиска (словари, тезаурусы, списки синонимов и стоп-слов) из старого кластера в новый.

6. **Настроить аутентификацию**

   Программа `pg_upgrade` будет подключаться к новому и старому серверу несколько раз, так что имеет смысл установить режим аутентификации `peer` в `pg_hba.conf` или использовать файл `~/.pgpass` (см. Раздел 31.15).

7. **Остановить оба сервера**

   Убедитесь в том, что оба сервера баз данных остановлены. Для этого в Unix можно выполнить:

   ```
   pg_ctl -D /opt/PostgreSQL/9.6 stop
   pg_ctl -D /opt/PostgreSQL/10 stop
   ```

   А в Windows, с соответствующими именами служб:

   ```
   NET STOP postgresql-9.6
   NET STOP postgresql-10
   ```

   Ведомые серверы с потоковой репликацией и трансляцией журнала могут продолжать работать до последнего шага.

8. **Подготовиться к обновлению ведомых серверов**

   Если вы производите обновление ведомых серверов (как описано в разделе Шаг 10), удостоверьтесь, что эти серверы находятся в актуальном состоянии, запустив pg_controldata в старых ведущем и ведомых кластерах. Убедитесь в том, что «Положение последней контрольной точки» во всех кластерах одинаковое. (Несовпадение будет иметь место, если старые ведомые серверы будут отключены раньше, чем старый ведущий, или если старые ведомые серверы всё ещё продолжают работать.) Также убедитесь в том, что в файле `postgresql.conf` нового ведущего кластера значение `wal_level` выше `minimal`.

9. **Запустить pg_upgrade**

   Всегда запускайте программу pg_upgrade от нового сервера, а не от старого. pg_upgrade требует указания каталогов данных старого и нового кластера, а также каталогов исполняемых файлов (`bin`). Вы можете также определить имя пользователя и номера портов, и нужно ли копировать файлы данных (по умолчанию) или создавать ссылки на них.

   Если выбрать вариант со ссылкой на данные, обновление выполнится гораздо быстрее (так как файлы не копируются) и потребует меньше места на диске, но вы лишитесь возможности обращаться к вашему старому кластеру, запустив новый после обновления. Этот вариант также требует, чтобы каталоги данных старого и нового кластера располагались в одной файловой системе. (Табличные пространства и `pg_wal` могут находиться в других файловых системах.) Полный список параметров вы можете получить, выполнив `pg_upgrade --help`.

   Параметр `--jobs` позволяет задействовать для копирования/связывания файлов и для выгрузки/перезагрузки схем баз данных несколько процессорных ядер. В качестве начального значения параметра стоит выбрать максимум из числа процессорных ядер и числа табличных пространств. Этот параметр может радикально сократить время обновления сервера со множеством баз данных, работающего в многопроцессорной системе.

   В Windows вы должны войти в систему с административными полномочиями, затем запустить командную строку от имени пользователя `postgres`, задать подходящий путь:

```
RUNAS /USER:postgres "CMD.EXE"
SET PATH=%PATH%;C:\Program Files\PostgreSQL\10\bin;
```

   Наконец, запустить pg_upgrade с путями каталогов в кавычках, например, так:

```
pg_upgrade.exe
        --old-datadir "C:/Program Files/PostgreSQL/9.6/data"
        --new-datadir "C:/Program Files/PostgreSQL/10/data"
        --old-bindir "C:/Program Files/PostgreSQL/9.6/bin"
        --new-bindir "C:/Program Files/PostgreSQL/10/bin"
```

   При запуске `pg_upgrade` проверит два кластера на совместимость и, если всё в порядке, выполнит обновление. Также возможно запустить `pg_upgrade --check`, чтобы ограничиться только проверками (при этом старый сервер может продолжать работать). Команда `pg_upgrade --check` также сообщит, какие коррективы вам нужно будет внести вручную после обновления. Если вы планируете использовать режим ссылок на данные, укажите вместе с `--check` параметр `--link`, чтобы были проведены специальные проверки для этого режима. Программе `pg_upgrade` требуются права на запись в текущий каталог.

   Очевидно, никто не должен обращаться к кластерам в процессе обновления. Программа pg_upgrade по умолчанию запускает серверы с портом 50432, чтобы не допустить нежелательных клиентских подключений. В процессе обновления оба кластера могут использовать один номер порта, так как они не будут работать одновременно. Однако для проверки старого работающего сервера новый порт должен отличаться от старого.

   Если при восстановлении схемы базы данных происходит ошибка, `pg_upgrade` завершает свою работу и вы должны вернуться к старому кластеру, как описывается ниже в Шаг 16. Чтобы попробовать `pg_upgrade` ещё раз, вы должны внести коррективы в старом кластере, чтобы pg_upgrade могла успешно восстановить схему. Если проблема возникла в модуле `contrib`, может потребоваться удалить этот модуль `contrib` в старом кластере, а затем установить его в новом после обновления (предполагается, что этот модуль не хранит пользовательские данные).

10. **Обновить ведомые серверы с потоковой репликацией и трансляцией журнала**

    Если вы используете режим ссылок и у вас реализована потоковая репликация (см. Подраздел 25.2.5) или трансляция журнала (см. Раздел 25.2) для ведомых серверов, вы можете быстро обновить эти серверы следующим образом. Вам не нужно будет запускать на них

pg_upgrade, вместо этого вы выполните rsync на ведущем. Не запускайте никакие серверы на этом этапе.

Если вы *не* используете режим ссылок, либо у вас нет rsync или вы не хотите его использовать, либо если вам нужен более простой вариант, пропустите инструкции в этом разделе и просто пересоздайте ведомые серверы сразу после завершения pg_upgrade и запуска нового ведущего сервера.

a. **Установите новые исполняемые файлы Postgres Pro на ведомых серверах**

   Убедитесь в том, что на всех ведомых серверах установлены новые исполняемые и вспомогательные файлы.

b. **Убедитесь в том, что новые каталоги данных на ведомых серверах *не* существуют**

   Новые каталоги данных ведомых серверов должны *отсутствовать* либо быть пустыми. Если запускалась программа initdb, удалите новые каталоги данных на ведомых.

c. **Установить разделяемые объектные файлы расширения**

   Установите на новых ведомых серверах те же разделяемые объектные файлы расширения, что вы установили в новом ведущем кластере.

d. **Остановите ведомые серверы**

   Если ведомые серверы продолжают работу, остановите их, следуя приведённым выше инструкциям.

e. **Сохраните файлы конфигурации**

   Сохраните все нужные вам файлы конфигурации из старых каталогов конфигурации ведомых серверов, в частности `postgresql.conf` (и все файлы, включённые в него), `postgresql.auto.conf`, `recovery.conf` и `pg_hba.conf`, так как они будут перезаписаны или удалены на следующем этапе.

f. **Запустите rsync**

   Когда используется режим ссылок, ведомые серверы можно быстро обновить, применив rsync. Для этого в каталоге, внутри которого находятся каталоги старого и нового кластера, для каждого ведомого сервера выполните на *ведущем*:

   ```
   rsync --archive --delete --hard-links --size-only --no-inc-recursive old_cluster
    new_cluster remote_dir
   ```

   Здесь каталоги `old_cluster` и `new_cluster` задаются относительно текущего каталога на ведущем, а `remote_dir` находится *над* каталогами старого и нового кластера на ведомом. Структура подкаталогов в заданных каталогах на ведущем и ведомых серверах должна быть одинаковой. Обратитесь к странице руководства rsync, где подробно описано, как указать удалённый каталог, например так:

   ```
   rsync --archive --delete --hard-links --size-only --no-inc-recursive /opt/
   PostgreSQL/9.5 \
         /opt/PostgreSQL/9.6 standby.example.com:/opt/PostgreSQL
   ```

   Проверить, что будет делать команда, можно, воспользовавшись параметром rsync `--dry-run`. Выполнить rsync на ведущем необходимо как минимум с одним ведомым, но затем, пока обновлённый ведомый остаётся остановленным, можно запускать rsync на нём для обновления других ведомых.

   В ходе этой операции записываются ссылки, созданные режимом ссылок pg_upgrade, связывающие файлы нового и старого кластера на ведущем сервере. Затем в старом кластере ведомого находятся соответствующие файлы и в новом кластере ведомого создаются ссылки на них. Файлы, не связанные ссылками на ведущем, копируются с него на ведомый. (Обычно их объём невелик.) Это позволяет произвести обновление ведомого быстро. К сожалению, при этом rsync будет напрасно копировать файлы, связанные с

временными и нежурналируемыми таблицами, так как они обычно не будут существовать на ведомых серверах.

Если у вас есть табличные пространства, вам потребуется выполнить подобную команду rsync для каталогов всех табличных пространств, например:

```
rsync --archive --delete --hard-links --size-only --no-inc-recursive /vol1/
pg_tblsp/PG_9.5_201510051 \
      /vol1/pg_tblsp/PG_9.6_201608131 standby.example.com:/vol1/pg_tblsp
```

Если вы вынесли `pg_wal` за пределы каталогов данных, нужно будет запустить rsync и для этих каталогов.

g. **Настройте ведомые серверы с потоковой репликацией и трансляцией журнала**

Настройте серверы для трансляции журнала. (Запускать `pg_start_backup()` и `pg_stop_backup()` или делать копию файловой системы не нужно, так как ведомые серверы остаются синхронизированными с ведущим.) Слоты репликации не копируются, их нужно создавать заново.

11. **Восстановить `pg_hba.conf`**

Если вы изменяли `pg_hba.conf`, восстановите его исходное состояние. Также может потребоваться скорректировать другие файлы конфигурации в новом кластере, чтобы они соответствовали старому, например, `postgresql.conf` (и файлы, включённые в него) и `postgresql.auto.conf`.

12. **Запустить новый сервер**

Теперь можно безопасно запустить новый сервер, а затем ведомые серверы, синхронизированные с ним с помощью rsync.

13. **Действия после обновления**

Если после обновления требуются какие-то дополнительные действия, программа pg_upgrade выдаст предупреждения об этом по завершении работы. Она также сгенерирует файлы скриптов, которые должны запускаться администратором. Эти скрипты будут подключаться к каждой базе данных, требующей дополнительных операций. Каждый такой скрипт следует выполнять командой:

```
psql --username=postgres --file=script.sql postgres
```

Эти скрипты могут выполняться в любом порядке, а после выполнения их можно удалить.

> **Внимание**
>
> Обычно к таблицам, задействованным в перестраивающих базу скриптах, опасно обращаться, пока эти скрипты не сделают свою работу; при этом можно получить некорректный результат или плохую производительность. К таблицам, не задействованным в таких скриптах, можно обращаться немедленно.

14. **Статистика**

Так как статистика оптимизатора не передаётся в процессе работы `pg_upgrade`, вы получите указание запустить соответствующую команду для воссоздания этой информации после обновления. Возможно, для этого вам понадобится установить параметры подключения к новому кластеру.

15. **Удалить старый кластер**

Если вы удовлетворены результатами обновления, вы можете удалить каталоги данных старого кластера, запустив скрипт, упомянутый в выводе `pg_upgrade` после обновления. (Автоматическое удаление невозможно, если в старом каталоге данных находятся дополнительные табличные пространства.) Также вы можете удалить каталоги старой инсталляции (например, `bin`, `share`).

16. **Возврат к старому кластеру**

    Если выполнив команду `pg_upgrade`, вы захотите вернуться к старому кластеру, возможны следующие варианты:

    - Если использовался ключ `--check`, в старом кластере ничего не меняется; его можно просто перезапустить.
    - Если *не* использовался ключ `--link`, в старом кластере ничего не меняется; его можно просто перезапустить.
    - Если использовался ключ `--link`, у старого и нового кластера могут оказаться общие файлы данных:
        - Если работа `pg_upgrade` была прервана до начала расстановки ссылок, в старом кластере ничего не меняется; его можно просто перезапустить.
        - Если вы *не* запускали новый кластер, старый кластер не претерпел никаких изменений, за исключением того, что при создании ссылки на данные к имени `$PGDATA/global/pg_control` было добавлено окончание `.old`. Чтобы продолжить использование старого кластера, достаточно убрать окончание `.old` из имени файла `$PGDATA/global/pg_control`; после этого старый кластер можно будет перезапустить.
        - Если вы запускали новый кластер, он внёс изменения в общие файлы, и использовать старый кластер небезопасно. В этом случае старый кластер нужно будет восстановить из резервной копии.

Замечания

pg_upgrade не поддерживает обновление баз данных, содержащих следующие ссылающиеся на OID системные типы данных `reg*`: `regproc`, `regprocedure`, `regoper`, `regoperator`, `regconfig` и `regdictionary`. (Обновление `regtype` возможно.)

Программа pg_upgrade сообщит обо всех актуальных для вашей инсталляции ошибках и потребностях перестроения или переиндексации базы; при этом будут созданы завершающие обновление скрипты, перестраивающие таблицы или индексы. Если вы попытаетесь автоматизировать обновление множества серверов, вы обнаружите, что для кластеров с одинаковыми схемами баз данных потребуются одинаковые действия после обновления; это объясняется тем, что эти действия диктуются схемой базы данных, а не данными пользователей.

Для проверки развёртывания новой версии создайте копию только схемы старого кластера, наполните этот кластер фиктивными данными, и попробуйте обновить его.

Если вы производите обновление кластера PostgreSQL версии до 9.2, в которой используется каталог только с файлами конфигурации, вы должны передать расположение собственно каталога с данными программе pg_upgrade, а расположение каталога конфигурации передать серверу, например `-d /каталог-данных -o '-D /каталог-конфигурации'`.

Если вы используете старый сервер версии до 9.1, работающий с нестандартным каталогом Unix-сокетов, либо его стандартное расположение отличается от принятого в новой версии, задайте в `PGHOST` расположение сокета старого сервера. (К Windows это не относится.)

При обновлении экземпляра Postgres Pro версии 9.6 или старее для баз данных с многобайтовой кодировкой утилита pg_upgrade может сгенерировать файлы SQL с командами `REINDEX`/`VALIDATE`. Вы должны запустить эти файлы, чтобы перестроить индексы или перепроверить ограничения. Эти файлы могут быть созданы в следующих случаях:

- В старом кластере используются индексы или ограничения, которые зависят от правил сортировки, отличных от основного правила сортировки БД, `C`, или `POSIX`.
- В Windows в старом кластере используются индексы или ограничения, которые зависят от основного правила сортировки с развёрнутым именем, например `"Russian_Russia[.`*кодировка*`]"` или `"English_United States[.`*кодировка*`]"`.

Если вы хотите использовать режим ссылок на данные, но не хотите каких-либо изменений в старом кластере при запуске нового, сделайте копию старого кластера и обновите его в этом режиме. Чтобы получить рабочую копию старого кластера, воспользуйтесь командой `rsync` и создайте предварительную копию кластера при работающем сервере, а затем отключите старый сервер и ещё раз запустите `rsync --checksum`, чтобы привести эту копию в согласованное состояние. (Ключ `--checksum` необходим, потому что `rsync` различает время с точностью только до секунд.) При этом вы можете исключить некоторые файлы, например `postmaster.pid`, как описано в Подразделе 24.3.3. Если ваша файловая система поддерживает снимки файловой системы или копирование при записи, вы можете воспользоваться этим для создания копии старого кластера и табличных пространств; при этом важно, чтобы такие снимки и копии файлов создавались одномоментно или когда сервер баз данных отключён.

См. также

initdb, pg_ctl, pg_dump, postgres

pg_waldump

pg_waldump — вывести журнал предзаписи кластера БД Postgres Pro в понятном человеку виде

Синтаксис

`pg_waldump` [`параметр`...] [`начальный_сегмент` [`конечный_сегмент`]]

Описание

Программа `pg_waldump` показывает содержимое журнала предзаписи (WAL) и прежде всего полезна для целей отладки и обучения.

Эту утилиту может запускать только пользователь, установивший сервер, так как ей требуется доступ на чтение к каталогу данных.

Параметры

Следующие аргументы командной строки задают расположение данных и формат вывода:

`начальный_сегмент`

Начать чтение с указанного файла сегмента журнала. Это неявно определяет каталог, в котором будут находиться файлы, и целевую линию времени.

`конечный_сегмент`

Остановиться после чтения указанного файла сегмента журнала.

`-b`
`--bkp-details`

Выводить подробные сведения о блоках-копиях страниц.

`-e` *`конец`*
`--end=`*`конец`*

Прекратить чтение в заданной позиции в WAL, а не читать поток до конца.

`-f`
`--follow`

Достигнув конца корректного WAL, проверять раз в секунду поступление новых записей WAL.

`-n` *`предел`*
`--limit=`*`предел`*

Вывести заданное число записей и остановиться.

`-p` *`путь`*
`--path=`*`путь`*

Задаёт каталог, содержащий файлы сегментов журнала, либо каталог с подкаталогом `pg_wal`, содержащим такие файлы. По умолчанию в поисках этих файлов просматривается текущий каталог, подкаталог `pg_wal` текущего каталога и подкаталог `pg_wal` каталога `PGDATA`.

`-r` *`менеджер_ресурсов`*
`--rmgr=`*`менеджер_ресурсов`*

Выводить только записи, созданные указанным менеджером ресурсов. Когда в качестве имени менеджера передаётся `list`, программа выводит только список возможных имён менеджеров ресурсов и завершается.

`-s` *начало*
`--start=`*начало*

Позиция в WAL, с которой нужно начать чтение. По умолчанию чтение начинается с первой корректной записи журнала в самом первом из найденных файлов.

`-t` *линия_времени*
`--timeline=`*линия_времени*

Линия времени, из которой будут читаться записи журнала. По умолчанию используется значение, заданное параметром *начальный_сегмент*, если он присутствует, а иначе — 1.

`-V`
`--version`

Вывести версию pg_waldump и завершиться.

`-x` *ид_транзакции*
`--xid=`*ид_транзакции*

Вывести только записи, относящиеся к указанной транзакции.

`-z`
`--stats[=record]`

Вывести общую статистику (число и размер записей и образов полных страниц) вместо отдельных записей. Возможен вариант получения статистики по записям, а не по менеджерам ресурсов.

`-?`
`--help`

Вывести справку об аргументах командной строки pg_waldump и завершиться.

Замечания

Когда сервер работает, результаты могут быть некорректными.

Выводятся записи только указанной линии времени (или линии времени по умолчанию, если она не задана явно). Записи в других линиях времени игнорируются.

pg_waldump не будет читать файлы WAL с расширением `.partial`. Если требуется прочитать такие файлы, расширение `.partial` нужно убрать из их имён.

См. также

Раздел 29.5

postgres

postgres — Сервер баз данных Postgres Pro

Синтаксис

`postgres` [*параметр*...]

Описание

`postgres` это сервер баз данных Postgres Pro. Для получения доступа к базе данных клиент устанавливает соединение (локально или по сети) с сервером `postgres`. После установки соединения сервер `postgres` поднимает выделенный процесс для его обслуживания.

Один экземпляр `postgres` всегда управляет данными ровно одного кластера баз данных. Кластер — это коллекция баз данных, хранящихся в файловой системе в определённом размещении («области данных»). На одном физическом сервере можно запустить несколько экземпляров `postgres` одновременно, при условии, что они используют различные области данных и порты. При запуске `postgres` необходимо указать размещение данных, которое задаётся в параметре `-D` или переменной окружения `PGDATA`, значение по умолчанию отсутствует. Обычно `-D` или `PGDATA` указывает на каталог, созданный во время развёртывания кластера с помощью initdb. Иные варианты рассмотрены в Разделе 18.2.

По умолчанию `postgres` запускается не в фоновом режиме, а вывод журнала осуществляет в стандартный поток ошибок. На практике `postgres` должен запускаться в фоновом режиме, возможно, при старте системы.

Команду `postgres` также возможно использовать в однопользовательском режиме. В основном этот режим используется на этапе инициализации при выполнении initdb. Иногда он также применяется в целях отладки или после аварийного сбоя. Заметьте, что однопользовательский режим не вполне подходит для отладки сервера ввиду отсутствия в нём межпроцессного взаимодействия и блокировок. Когда сервер запускается в однопользовательском режиме из командной строки, он может принимать запросы и выводить их результаты на экран, но формат этого вывода ориентирован больше на разработчиков, чем на обычных пользователей. В этом режиме текущим пользователем считается пользователь под номером 1, который неявно наделяется правами суперпользователя. При этом данный пользователь может фактически не существовать, поэтому в ряде случаев этот режим даёт возможность вручную восстановить базу при повреждении системных каталогов.

Параметры

Программа `postgres` принимает следующие аргументы командной строки. За подробным описанием параметров обратитесь к Главе 18. От необходимости вводить большинство этих параметров можно избавиться, записав их в файл конфигурации. Некоторые (безопасные) параметры можно также задать со стороны подключающегося клиента (в зависимости от приложения), чтобы они применялись только к одному сеансу. Например, если установлена переменная окружения `PGOPTIONS`, клиенты на базе libpq передадут эту строку серверу, который воспримет её как параметры, передаваемые в командной строке `postgres`.

Параметры общего назначения

`-B` *количество-буферов*

Устанавливает количество разделяемых между процессами буферов. Значение по умолчанию выбирается автоматически при развёртывании кластера с помощью initdb. Установка флага аналогична конфигурации параметра shared_buffers.

`-c` *имя=значение*

Устанавливает заданный параметр времени выполнения. Конфигурационные параметры, поддерживаемые Postgres Pro, описаны в Главе 18. Большинство других параметров командной

строки на самом деле представляют собой краткие формы такого присваивания значений параметрам. Для установления нескольких параметров `-c` можно указывать многократно.

`-C` *имя*

Отображает значение заданного параметра времени выполнения и завершается (подробнее см. выше). Можно применять на работающем сервере, при этом будут возвращены значения `postgresql.conf` с учётом проведённых в рамках вызова изменений. Значения, переданные при старте кластера, не отображаются.

Параметр предназначен для приложений, взаимодействующих с сервером, например pg_ctl, и запрашивающих параметры конфигурации. Пользовательские приложения должны использовать команду SHOW или представление `pg_settings`.

`-d` *уровень-отладки*

Устанавливает уровень отладки (от 1 до 5). Чем выше значение, тем подробнее осуществляется вывод в журнал сервера. Также возможно передать `-d 0` для отдельной сессии, что предотвратит в её рамках влияние выставленного для `postgres` значения.

`-D` *datadir*

Указывает размещение конфигурационных файлов базы в пределах файловой системы. За подробностями обратитесь к Разделу 18.2.

`-e`

Устанавливает формат вводимых дат по умолчанию в «European» с последовательностью значений `DMY`. Также влияет на вывод дня, идущего перед значением месяца, более подробно см. Раздел 8.5.

`-F`

Отключает вызовы `fsync` для увеличения производительности, но с увеличением рисков потери данных в случае краха системы. Этот параметр работает аналогично параметру конфигурации fsync. Внимательно прочтите документацию перед использованием данного параметра!

`-h` *компьютер*

Указывает IP-адрес или имя компьютера, на котором сервер `postgres` принимает клиентские подключения по TCP/IP. Значением может быть список адресов, разделённых запятыми, либо символ `*`, обозначающий все доступные интерфейсы. Если значение опущено, то подключения принимаются только через Unix-сокеты. По умолчанию принимаются подключения только к localhost. Флаг работает аналогично конфигурационному параметру listen_addresses.

`-i`

Позволяет клиентам подключаться по TCP/IP. Без этого параметра допускаются лишь локальные подключения. Действие этого параметра аналогично действию параметра конфигурации `listen_addresses` со значением `*` в `postgresql.conf` или ключа `-h`.

Параметр устарел, так как не даёт полной функциональности listen_addresses. Лучше устанавливать значение `listen_addresses` напрямую.

`-k` *каталог*

Указывает каталог Unix-сокета, через который `postgres` будет принимать подключения. Значением параметра может быть список каталогов через запятую. Если это значение пустое, использование Unix-сокетов запрещается, разрешаются только подключения по TCP/IP. По умолчанию выбирается каталог `/tmp`, но его можно сменить на этапе компиляции. Этот параметр действует аналогично параметру конфигурации unix_socket_directories.

`-l`

Включает поддержку безопасных соединений с использованием SSL шифрования. Postgres Pro необходимо скомпилировать с поддержкой SSL для использования этого флага. Подробнее использование SSL описано в Раздел 17.9.

`-N` *максимальное количество соединений*

Устанавливает максимально возможное количество одновременных клиентских соединений. Значение по умолчанию устанавливается автоматически на этапе развёртывания с помощью initdb. Флаг работает аналогично конфигурационному параметру max_connections.

`-o` *дополнительные-параметры*

Аргументы командной строки, поступившие через *дополнительные-параметры*, передаются всем обслуживающим процессам, запускаемым этим процессом `postgres`.

Пробелы в строке *дополнительные-параметры* воспринимаются как разделяющие аргументы, если перед ними нет обратной косой черты (`\`); чтобы представить обратную косую черту буквально, продублируйте её (`\\`). Также можно задать несколько аргументов, используя `-o` несколько раз.

Использование этого параметра считается устаревшим, так как на данный момент все параметры `postgres` можно задать в командной строке.

`-p` *порт*

Указывает порт TCP/IP или расширение файла локального Unix-сокета, через который `postgres` принимает подключения клиентских приложений. По умолчанию принимает значение переменной окружения `PGPORT`, или, если значение `PGPORT` не установлено, то используется значение, установленное на этапе компиляции (обычно это 5432). Если значение порта меняется, то на стороне клиентов это необходимо учитывать, установив, либо `PGPORT`, либо флаг командной строки.

`-s`

Отображает информацию о времени и другую статистику после каждой выполненной команды, что полезно для оценки производительности во время настройки количества буферов.

`-S` *рабочая-память*

Указывает объём памяти, который сервер будет использовать для внутренних операций сортировки и хеширования, прежде чем прибегнуть к использованию временных файлов на диске. Обратитесь к описанию параметра `work_mem`, приведённому в Подразделе 18.4.1.

`-V`
`--version`

Отображает версию postgres и прерывает дальнейшее выполнение.

`--`*имя*`=`*значение*

Устанавливает заданный параметр времени выполнения. Является короткой формой ключа `-c`.

`--describe-config`

Выводит значения конфигурационных переменных сервера, их описаний и значений по умолчанию в формате команды `COPY` со знаком табуляции в качестве разделителя. В основном это предназначено для средств администрирования.

`-?`
`--help`

Выводит помощь по аргументам команды postgres.

Параметры для внутреннего использования

Далее описанные параметры, в основном, применяются в целях отладки, а в некоторых случаях при восстановлении сильно повреждённых баз данных. Их описание приведено для системных разработчиков Postgres Pro, поэтому они могут быть изменены без уведомления.

`-f { s | i | o | b | t | n | m | h }`

Запрещает использование специфических методов сканирования и объединения: `s` и `i` выключают последовательное сканирование и по индексу соответственно, а `o`, `b` и `t` выключает сканирование только по индексу, сканирование по битовым векторам, и сканирование по ID кортежей соответственно, в то время как `n`, `m` и `h` выключает вложенные циклы, слияния и хеширование соответственно.

Ни последовательное сканирование, ни вложенные циклы невозможно выключить полностью. Флаги `-fs` и `-fn` просто указывают планировщику избегать выполнения этих операций при наличии других альтернатив.

`-n`

Параметр предназначен для отладки сервера в случае аномального завершения процесса. Обычная практика в таком случае — завершение порождённых процессов с дальнейшей инициализацией разделяемой памяти и семафоров. Это связано с тем, что потерянный процесс мог повредить область разделяемой памяти. Параметр указывает `postgres` не производить повторной инициализации общих структур данных, что позволяет произвести дальнейшую отладку текущего состояния памяти и семафоров.

`-O`

Разрешает модифицировать структуру системных таблиц. Используется командой `initdb`.

`-P`

Игнорировать системные индексы при чтении, но продолжать обновлять их при изменениях системных таблиц. Это используется при их восстановлении после повреждения.

`-t pa[rser] | pl[anner] | e[xecutor]`

Выводит статистику по времени исполнения каждого запроса в контексте каждого системного модуля. Использование флага совместно с `-s` невозможно.

`-T`

Параметр предназначен для отладки сервера в случае аномального завершения процесса. Обычная практика в таком случае — завершение порождённых процессов с дальнейшей инициализацией разделяемой памяти и семафоров. Это связано с тем, что потерянный процесс мог повредить область разделяемой памяти. Параметр указывает `postgres` на необходимость остановки порождённых процессов сигналом `SIGSTOP`, но не завершит их, что позволяет разработчикам сделать снимки памяти процессов.

`-v` *протокол*

Указывает для данного сеанса версию протокола взаимодействия сервера с клиентом. Флаг используется лишь для внутренних целей.

`-W` *секунды*

При старте сервера производится задержка на указанное количество секунд, после чего производится процедура аутентификации, что позволяет подключить отладчик к процессу.

Параметры для однопользовательского режима

Следующие параметры применимы только для однопользовательского режима (см. Раздел «Однопользовательский режим»).

`--single`

Устанавливает однопользовательский режим. Параметр должен идти первым в командной строке.

`база_данных`

Указывает имя базы данных, к которой производится подключение. Параметр должен идти последним в командной строке. Если не указан, то используется имя текущего системного пользователя.

`-E`

Выводить все команды в устройство стандартного вывода прежде чем выполнять их.

`-j`

Считать признаком окончания ввода команды точку с запятой с двумя переводами строки, а не один перевод строки.

`-r` *`имя_файла`*

Отправляет вывод журнала сервера в файл *`filename`*. Этот параметр применяется лишь при запуске из командной строки.

Переменные окружения

`PGCLIENTENCODING`

Кодировка, используемая клиентом по умолчанию. Может переопределяться на стороне клиента, а также устанавливаться в конфигурационном файле сервера.

`PGDATA`

Каталог размещения данных кластера по умолчанию

`PGDATESTYLE`

Значение по умолчанию для параметра времени выполнения DateStyle. Применение этой переменной является устаревшим.

`PGPORT`

Порт по умолчанию, лучше устанавливать в конфигурационном файле.

Диагностика

Ошибки с упоминанием о `semget` или `shmget` говорят о возможной необходимости проведения более оптимального конфигурирования ядра. Подробнее это обсуждается в Разделе 17.4. Отложить переконфигурирование можно, уменьшив shared_buffers для снижения общего потребления разделяемой памяти Postgres Pro и/или уменьшив max_connections для снижения затрат на использование семафоров.

Необходимо внимательно проверять сообщения об ошибке с упоминанием о другом запущенном экземпляре, например, с использованием команды

```
$ ps ax | grep postgres
```

или

```
$ ps -ef | grep postgres
```

в зависимости от ОС. Если есть полная уверенность, что противоречий нет, необходимо самостоятельно удалить упомянутый в сообщении запирающий файл и повторить попытку.

Упоминание о невозможности привязки к порту в сообщениях об ошибках может указывать на то, что он уже занят другим процессом помимо Postgres Pro. Также сообщение может возникнуть

при мгновенном рестарте postgres на том же порту. В этом случае нужно немного подождать, пока ОС не закроет порт, и повторить попытку. Ещё возможна ситуация, в которой используется резервный системный порт. Например, многие Unix-подобные ОС резервируют «доверительные» порты от 1024 и ниже, и лишь суперпользователь имеет к ним доступ.

Замечания

Для комфортного запуска и остановки сервера можно использовать утилиту pg_ctl.

Если возможно, *не используйте* сигнал SIGKILL для головного процесса postgres. В этом случае postgres не освободит системные ресурсы, например, разделяемую память и семафоры. Это может привести к проблемам при повторном запуске postgres.

Для корректного завершения postgres используются сигналы SIGTERM, SIGINT или SIGQUIT. При первом будут ожидаться все дочерние процессы до их завершения, второй приведёт к принудительному закрытию соединений, а третий — к незамедлительному выходу без корректного завершения, приводящему к необходимости выполнения процедуры восстановления на следующем старте.

Получая сигнал SIGHUP, сервер перечитывает свои файлы конфигурации. Также возможно отправить SIGHUP отдельному процессу, но это чаще всего бессмысленно.

Для отмены исполняющегося запроса, отправьте SIGINT обслуживающему его процессу. Для чистого завершения серверного процесса отправьте ему SIGTERM. Также см. pg_cancel_backend и pg_terminate_backend в Подразделе 9.26.2, которые являются аналогами в форме SQL-инструкций.

Сервер postgres обрабатывает SIGQUIT для завершения дочерних процессов в грязную, и сигнал *не должен* отправляться пользователем. Также не стоит посылать SIGKILL серверному процессу — головной postgres процесс расценит это как аварию и принудительно завершит остальные порождённые, как это было бы сделано при процедуре восстановления после сбоя.

Ошибки

Флаги, начинающиеся с -- не работают в ОС FreeBSD или OpenBSD. Чтобы обойти это, используйте -c. Это ошибка ОС. В будущих релизах Postgres Pro будет предоставлен обходной путь, если ошибка так и не будет устранена.

Однопользовательский режим

Для запуска сервера в однопользовательском режиме используется, например, команда

postgres --single -D /usr/local/pgsql/data *другие параметры* my_database

Необходимо указать корректный путь к каталогу хранения данных в параметре -D, или установить переменную окружения PGDATA. Также замените имя базы данных на необходимое.

Обычно перевод строки в однопользовательском режиме сервер воспринимает как завершение ввода команды; точка с запятой не имеет для него такого значения, как для psql. Поэтому, чтобы ввести команду, занимающую несколько строк, необходимо добавлять в конце каждой строки, кроме последней, обратную косую черту. Обратная косая черта и следующий за ней символ перевода строки автоматически убираются из вводимой команды. Заметьте, что это происходит даже внутри строковой константы или комментария.

Но если применить аргумент командной строки -j, одиночный символ перевода строки не будет завершать ввод команды; это будет делать последовательность «точка с запятой, перевод строки, перевод строки». То есть для завершения команды нужно ввести точку с запятой, и сразу за ней пустую строку. Просто точка с запятой, дополненная переводом строки, в этом режиме не имеет специального значения. Внутри строковых констант и комментариев такая завершающая последовательность воспринимается в том же ключе.

Вне зависимости от режима ввода, символ точки с запятой, введённый не прямо перед или в составе последовательности завершения команды, воспринимается как разделитель команд. После ввода завершающей последовательности введённые с разделителями несколько операторов будут выполняться в одной транзакции.

Для завершения сеанса введите символ конца файла (EOF, обычно это сочетание **Control+D**). Если вы вводили текст после окончания ввода последней команды, символ EOF будет воспринят как символ завершения команды, и для выхода потребуется ещё один EOF.

Заметьте, что однопользовательский режим не предоставляет особых возможностей для редактирования команд (например, нет истории команд). Также в однопользовательском режиме не производятся никакие фоновые действия, например, не выполняются автоматические контрольные точки или репликация.

Примеры

Для запуска `postgres` в фоновом режиме с параметрами по умолчанию:

```
$ nohup postgres >logfile 2>&1 </dev/null &
```

Для запуска `postgres` с определённым портом, например, 1234:

```
$ postgres -p 1234
```

Для соединения с помощью psql укажите этот порт в параметре -p:

```
$ psql -p 1234
```

или в переменной окружения `PGPORT`:

```
$ export PGPORT=1234
$ psql
```

Именованный параметр времени выполнения можно указать одним из приведённых способом:

```
$ postgres -c work_mem=1234
$ postgres --work-mem=1234
```

Любой из методов переопределяет значение `work_mem` конфигурации `postgresql.conf`. Символ подчёркивания в именах можно указать и в виде тире. Задавать параметры обычно (не считая кратковременных экспериментов) лучше в `postgresql.conf`, а не в аргументах командной строки.

См. также

initdb, pg_ctl

postmaster

postmaster — Сервер баз данных Postgres Pro

Синтаксис

`postmaster` [*параметр*...]

Описание

`postmaster` это устаревшее название `postgres`.

См. также

postgres

Часть VII. Внутреннее устройство

Содержит разнообразную информацию, полезную для разработчиков Postgres Pro.

Глава 48. Обзор внутреннего устройства Postgres Pro

> **Автор**
>
> Основой этой главы послужил материал дипломной работы sim98, написанной Стефаном Симковичем (Stefan Simkovics) в Венском техническом университете под руководством профессора Георга Готлоба (Georg Gottlob) и его ассистентки Катрин Сейр (Katrin Seyr).

В этой главе даётся обзор внутренней организации сервера Postgres Pro. Прочитав следующие разделы, вы получите представление о том, как обрабатывается запрос. Здесь мы не стремились подробно описывать внутренние операции Postgres Pro, так как это заняло бы слишком большой объём. Основная цель этой главы другая — помочь читателю понять общую последовательность действий, выполняемых сервером с момента получения запроса до момента выдачи результатов клиенту.

48.1. Путь запроса

Ниже мы кратко опишем этапы, которые проходит запрос для получения результата.

1. Прикладная программа устанавливает подключение к серверу Postgres Pro. Эта программа передаёт запрос на сервер и ждёт от него результатов.
2. На *этапе разбора запроса* сервер выполняет синтаксическую проверку запроса, переданного прикладной программой, и создаёт *дерево запроса*.
3. *Система правил* принимает дерево запроса, созданное на стадии разбора, и ищет в *системных каталогах правила* для применения к этому дереву. Обнаружив подходящие правила, она выполняет преобразования, заданные в *теле правил*.

   Одно из применений системы правил заключается в реализации *представлений*. Когда выполняется запрос к представлению (т. е. *виртуальной таблице*), система правил преобразует запрос пользователя в запрос, обращающийся не к представлению, а к *базовым таблицам* из *определения представления*.
4. *Планировщик/оптимизатор* принимает дерево запроса (возможно, переписанное) и создаёт *план запроса*, который будет передан *исполнителю*.

   Он выбирает план, сначала рассматривая все возможные варианты получения одного и того же результата. Например, если для обрабатываемого отношения создан индекс, прочитать отношение можно двумя способами. Во-первых, можно выполнить простое последовательное сканирование, а во-вторых, можно использовать индекс. Затем оценивается стоимость каждого варианта и выбирается самый дешёвый. Затем выбранный вариант разворачивается в полноценный план, который сможет использовать исполнитель.
5. Исполнитель рекурсивно проходит по *дереву плана* и получает строки тем способом, который указан в плане. Он сканирует отношения, обращаясь к *системе хранения*, выполняет *сортировку* и *соединения*, вычисляет *условия фильтра* и, наконец, возвращает полученные строки.

В следующих разделах мы более подробно рассмотрим каждый из этих этапов, чтобы дать представление о внутренних механизмах и структурах данных Postgres Pro.

48.2. Как устанавливаются соединения

Postgres Pro реализует простую клиент-серверную модель по схеме «процесс для пользователя». В такой схеме один *клиентский процесс* подключается к одному отдельному *серверному процессу*. Так как мы не знаем заранее, сколько подключений будет, нам нужен *главный процесс*,

который будет запускать новый процесс при каждом запросе подключения. Главный процесс называется `postgres` и принимает входящие подключения в заданном порту TCP/IP. Получив запрос на подключение, процесс `postgres` порождает новый серверный процесс. Серверные задачи взаимодействуют между собой через *семафоры* и *разделяемую память*, чтобы обеспечить целостность данных при одновременном обращении к ним.

Клиентским процессом может быть любая программа, которая понимает протокол Postgres Pro, описанный в Главе 50. Многие клиенты базируются на библиотеке libpq для языка C, но есть и другие независимые реализации этого протокола, например, драйвер JDBC для Java.

Установив подключение, клиентский процесс может передать запрос серверу. Запрос передаётся в обычном текстовом виде, клиент не занимается его анализом. Сервер разбирает запрос, строит *план выполнения*, выполняет его и возвращает полученные строки клиенту, передавая их через установленное подключение.

48.3. Этап разбора

Этап разбора разделяется на две части:

- *Разбор*, алгоритм которого описан в `gram.y` и `scan.l`, а программный код генерируется инструментами Unix bison и flex.
- *Преобразование*, в процессе которого модифицируются и дополняются структуры данных, полученные после разбора запроса.

48.3.1. Разбор

При разборе проверяется сначала синтаксис строки запроса (поступающей в виде неструктурированного текста). Если он правильный, строится *дерево запроса* и передаётся дальше, в противном случае возвращается ошибка. Лексический и синтаксический анализ реализован с применением хорошо известных средств Unix bison и flex.

Лексическая структура определяется в файле `scan.l` и описывает *идентификаторы*, *ключевые слова SQL* и т. д. Для каждого найденного ключевого слова или идентификатора генерируется *символ языка*, который затем передаётся синтаксическому анализатору.

Синтаксис языка определён в файле `gram.y` в виде набора *грамматических правил* и *действий*, которые должны выполняться при срабатывании правил. Для построения дерева разбора используется код действий (это действительно код на C).

Файл `scan.l` преобразуется в программу на C `scan.c` с помощью flex, а `gram.y` — в `gram.c` с помощью bison. После этих преобразований исполняемый код анализатора создаётся обычным компилятором C. Никогда не вносите коррективы в сгенерированные файлы C, так как они будут перезаписаны при следующем вызове flex или bison.

> **Примечание**
>
> Упомянутые преобразования и компиляция обычно производятся автоматически сборочными файлами *Makefile*, поставляемыми в составе дистрибутива Postgres Pro.

Подробное описание bison и грамматических правил в `gram.y` выходит за рамки данной главы. Узнать больше о flex и bison можно из книг и документации. Изучение грамматики, описанной в `gram.y`, следует начать со знакомства с bison, иначе будет трудно понять, что там происходит.

48.3.2. Преобразование

На этой стадии дерево разбора создаётся только с фиксированными знаниями о синтаксической структуре SQL. При его создании не просматриваются системные каталоги, что не даёт возможность понять конкретную семантику запрошенной операции. После этого выполняется

процедура преобразования, которая принимает дерево разбора от анализатора и выполняет семантический анализ, необходимый для понимания, к каким именно таблицам, функциям и операторам обращается запрос. Структура данных, которая создаётся для представления этой информации, называется *деревом запроса*.

Синтаксический разбор отделён от семантического анализа, потому что обращаться к системным каталогам можно только внутри транзакции, а начинать транзакцию сразу после получения строки с запросом нежелательно. Синтаксического разбора достаточно, чтобы распознать команды управления транзакциями (`BEGIN`, `ROLLBACK` и т. д.), поэтому их можно выполнить без дальнейшего анализа. Убедившись, что мы имеем дело с собственно запросом (например, `SELECT` или `UPDATE`), можно начинать транзакцию, если она ещё не начата. Только после этого можно переходить к процедуре преобразования.

Дерево запроса, создаваемое процедурой преобразования, по структуре во многом похоже на дерево разбора, но отличается во многих деталях. Например, узел `FuncCall` в дереве разбора представляет то, что по синтаксису похоже на вызов функции. Этот узел может быть преобразован в узел `FuncExpr` или `Aggref` в зависимости от того, какой (обычной или агрегатной) окажется функция с заданным именем. Кроме того, в дерево запроса добавляется информация о фактических типах данных столбцов и результатов выражений.

48.4. Система правил Postgres Pro

Postgres Pro поддерживает мощную *систему правил* для создания *представлений* и возможности *изменения представлений*. Система правил Postgres Pro претерпела две реализации:

- Первый вариант производил обработку на *уровне строк* и был внедрён глубоко в *исполнителе*. Этот обработчик правил вызывался при обращении к каждой отдельной строке. Эта реализация была ликвидирована в 1995 г., когда последний официальный выпуск Berkeley Postgres превратился в Postgres95.
- Во втором воплощении системы правил применили так называемое *переписывание запроса*. *Система переписывания* реализована в механизме, внедрённом между *анализатором* и *планировщиком/оптимизатором*. Этот механизм работает и сегодня.

Механизм переписывания запросов подробно обсуждается в Главе 38, так что здесь мы его не рассматриваем. Мы только отметим, что и на входе, и на выходе у него деревья запросов, то есть представление или уровень семантической детализации он не меняет. Переписывание запроса можно считать формой расширения макросов.

48.5. Планировщик/оптимизатор

Задача *планировщика/оптимизатора* — построить наилучший план выполнения. Определённый SQL-запрос (а значит, и дерево запроса) на самом деле можно выполнить самыми разными способами, при этом получая одни и те же результаты. Если это не требует больших вычислений, оптимизатор запросов будет перебирать все возможные варианты планов, чтобы в итоге выбрать тот, который должен выполниться быстрее остальных.

> **Примечание**
>
> В некоторых ситуациях рассмотрение всех возможных вариантов выполнения запросов занимает слишком много времени и памяти. В частности, это имеет место при выполнении запросов с большим количеством операций соединения. Поэтому, чтобы выбрать разумный (но не обязательно наилучший) план запроса за приемлемое время, Postgres Pro использует *генетический оптимизатор запросов* (см. Главу 55), когда количество соединений превышает некоторый предел (см. geqo_threshold).

Процедура поиска лучшего плана на самом деле работает со структурами данных, называемыми *путями*, которые представляют собой упрощённые схемы планов, содержащие минимум информации, необходимый планировщику для принятия решений. Когда наиболее выгодный план

выбран, строится полноценное *дерево плана*, которое и передаётся исполнителю. Оно описывает желаемый план выполнения достаточно подробно, чтобы исполнитель мог обработать его. В продолжении этого раздела мы будем считать, что планы и пути по сути одно и то же.

48.5.1. Выработка возможных планов

Сначала планировщик/оптимизатор вырабатывает планы для сканирования каждого отдельного отношения (таблицы), используемого в запросе. Множество возможных планов определяется в зависимости от наличия индексов в каждом отношении. Произвести последовательное сканирование отношения можно в любом случае, так что план последовательного сканирования создаётся всегда. Предположим, что для отношения создан индекс (например, индекс-B-дерево) и запрос содержит ограничение `отношение.атрибут ОПЕР константа`. Если окажется, что `отношение.атрибут` совпадает с ключом индекса-B-дерева и `ОПЕР` — один из операторов, входящих в *класс операторов* индекса, создаётся ещё один план, с использованием индекса-B-дерева для чтения отношения. Если находятся другие индексы, ключи которых соответствуют ограничениям запроса, могут добавиться и другие планы. Планы сканирования индекса также создаются для индексов, если их порядок сортировки соответствует предложению `ORDER BY` (если оно есть), или этот порядок может быть полезен для соединения слиянием (см. ниже).

Если в запросе требуется соединить два или несколько отношений, после того, как будут определены все подходящие планы сканирования отдельных отношений, рассматриваются планы соединения. При этом возможны три стратегии соединения:

- *соединение с вложенным циклом*: Правое отношение сканируется один раз для каждой строки, найденной в левом отношении. Эту стратегию легко реализовать, но она может быть очень трудоёмкой. (Однако если правое отношение можно сканировать по индексу, эта стратегия может быть удачной. Тогда значения из текущей строки левого отношения могут использоваться как ключи для сканирования по индексу справа.)
- *соединение слиянием*: Каждое отношение сортируется по атрибутам соединения до начала соединения. Затем два отношения сканируются параллельно и соответствующие строки, объединяясь, формируют строки соединения. Этот тип соединения более привлекательный, так как каждое отношение сканируется только один раз. Требуемый порядок сортировки можно получить, либо добавив явный этап сортировки, либо просканировав отношение в нужном порядке, используя индекс по ключу соединения.
- *соединение по хешу*: сначала сканируется правое отношение и формируется хеш-таблица, ключ в которой вычисляется по атрибутам соединения. Затем сканируется левое отношение и по тем же атрибутам в каждой строке вычисляется ключ для поиска в этой хеш-таблице соответствующих строк справа.

Когда в запросе задействованы более двух отношений, окончательный результат должен быть получен из дерева с узлами соединения, имеющими по два входа. Планировщик рассматривает все возможные последовательности соединения и выбирает самую выгодную.

Если число задействованных в запросе отношений меньше geqo_threshold, для поиска оптимальной последовательности соединений производится практически полный перебор. Планировщик отдаёт предпочтение соединениям между двумя отношениями, для которых есть соответствующее предложение соединения в условии `WHERE` (то есть, для которых находится ограничение вида `where табл1.атр1=табл2.атр2`). Пары соединения без подобного предложения рассматриваются, только если нет другого выбора, то есть когда для определённого отношения не находятся предложения соединения с каким-либо другим отношением. Планировщик рассматривает все возможные планы для каждой пары соединения и выбирает самый выгодный из них (по его оценке).

Если `geqo_threshold` превышается, последовательность соединений выбирается эвристическим путём, как описано в Главе 55. В остальном процесс планирования тот же.

Законченное дерево плана содержит узлы сканирования по индексу или последовательного сканирования базовых отношений, плюс узлы соединения с вложенным циклом, соединения слиянием или соединения по хешу (если требуется), плюс, возможно, узлы дополнительных

действий, например, сортировки или вычисления агрегатных функций. Большинство из этих узлов могут дополнительно производить *отбор* (отбрасывать строки, не удовлетворяющие заданному логическому условию) и *расчёты* (вычислять производный набор столбцов по значениям заданных столбцов, то есть вычислять скалярные выражения). Одна из задач планировщика — добавить условия отбора из предложения `WHERE` и вычисления требуемых выходных выражений к наиболее подходящим узлам дерева плана.

48.6. Исполнитель

Исполнитель принимает план, созданный планировщиком/исполнителем и обрабатывает его рекурсивно, чтобы получить требуемый набор строк. Обработка выполняется по конвейеру, с получением данных по требованию. При вызове любого узла плана он должен выдать очередную строку, либо сообщить, что выдача строк завершена.

В качестве более конкретного примера, давайте предположим, что верхним узлом плана оказался узел `MergeJoin`. Для того чтобы выполнить какое-либо соединение, необходимо выбрать две строки (одну из каждого вложенного плана). Поэтому исполнитель рекурсивно вызывает себя для обработки вложенных планов (он начинает с плана `левого дерева`). Новый верхний узел (верхний узел левого вложенного плана) может быть, например, узлом `Sort`, и тогда для получения входной строки снова требуется рекурсия. Дочерним узлом `Sort` может быть узел `SeqScan`, представляющий собственно чтение таблицы. В результате выполнения этого узла исполнитель выбирает одну строку из таблицы и возвращает её вызывающему узлу. Узел `Sort`, в свою очередь, будет продолжать вызывать дочерний узел, пока не получит все строки для сортировки. Когда строки закончатся (дочерний узел сообщит об этом, возвратив NULL вместо строки), узел `Sort` выполнит сортировку, и наконец сможет выдать свою первую строку, а именно строку первую по порядку сортировки. Остальные строки будут сохраняться в нём, чтобы он мог выдавать их по порядку при последующих вызовах.

Узел `MergeJoin` подобным образом затребует первую строку и у вложенного плана справа. Затем он сравнивает две строки и определяет, можно ли их соединить; если да, он возвращает соединённую строки вызывающему узлу. При следующем вызове, или немедленно, если он не может соединить текущую пару поступивших строк, он переходит к следующей строке в одном отношении или в другом (в зависимости от результата сравнения) и снова проверяет соответствие. В конце концов, данные в одном или другом вложенном плане заканчиваются и узел `MergeJoin` возвращает NULL, показывая тем самым, что другие строки соединения получить нельзя.

Сложные запросы могут содержать много уровней вложенности узлов плана, но общий подход тот же: каждый узел вычисляет и возвращает следующую полученную строку при очередном вызове. Каждый узел также должен производить отбор и расчёты, которые были назначены ему планировщиком.

Механизм исполнителя применяется для обработки всех четырёх основных типов SQL-запросов: `SELECT`, `INSERT`, `UPDATE` и `DELETE`. С `SELECT` код исполнителя верхнего уровня должен только выдать клиенту все строки, полученные от дерева плана запроса. Запросы `INSERT ... SELECT`, `UPDATE` и `DELETE` по сути выполняются как `SELECT` под специальным узлом `ModifyTable` на верхнем уровне плана.

Команда `INSERT ... SELECT` подаёт строки в узел `ModifyTable` для добавления в отношение. С `UPDATE` планировщик делает так, чтобы каждая вычисленная строка включала значения всех изменённых столбцов плюс *TID* (Tuple ID, идентификатор кортежа) исходной целевой строки; эти данные подаются в узел `ModifyTable`, который использует эту информацию, чтобы создать новую изменённую строку и пометить старую строку как удалённую. С `DELETE` план фактически возвращает только один столбец, TID, а узел `ModifyTable` использует значения TID, чтобы найти каждую целевую строку и пометить её как удалённую.

Простая команда `INSERT ... VALUES` создаёт тривиальное дерево плана, содержащее единственный узел `Result`, который вычисляет ровно одну строку результата и подаёт её в вышестоящий узел `ModifyTable` для добавления в отношение.

Глава 49. Системные каталоги

Системные каталоги — это место, где система управления реляционной базой данных хранит метаданные схемы, в частности информацию о таблицах и столбцах, а также служебные сведения. Системные каталоги Postgres Pro представляют собой обычные таблицы. Поэтому вы можете удалить и пересоздать их, добавить столбцы, изменить и добавить строки, т. е. разными способами вмешаться в работу системы. Обычно модифицировать системные каталоги вручную не следует, для всего этого, как правило, есть команды SQL. (Например, `CREATE DATABASE` вставляет строку в каталог `pg_database` — и фактически создаёт базу данных на диске.) Исключение составляют только особенные эзотерические операции, но многие из них со временем становятся выполнимыми посредством SQL-команд, так что потребность напрямую модифицировать системные каталоги постоянно уменьшается.

49.1. Обзор

В Таблице 49.1 перечислены системные каталоги. Подробное описание каждого каталога следует далее.

Большинство системных каталогов копируются из базы-шаблона при создании базы данных и затем принадлежат этой базе. Но некоторые каталоги физически разделяются всеми базами данных в кластере; это отмечено в их описаниях.

Таблица 49.1. Системные каталоги

Имя каталога	Предназначение
pg_aggregate	агрегатные функции
pg_am	индексные методы доступа
pg_amop	операторы методов доступа
pg_amproc	опорные процедуры методов доступа
pg_attrdef	значения столбцов по умолчанию
pg_attribute	столбцы таблиц («атрибуты»)
pg_authid	идентификаторы для авторизации (роли)
pg_auth_members	отношения членства для объектов авторизации
pg_cast	приведения (преобразования типов данных)
pg_class	таблицы, индексы, последовательности, представления («отношения»)
pg_collation	правила сортировки (параметры локали)
pg_constraint	ограничения-проверки, ограничения уникальности, ограничения первичного ключа и внешних ключей
pg_conversion	информация о перекодировках
pg_database	базы данных в этом кластере
pg_db_role_setting	параметры, задаваемые на уровне ролей и баз данных
pg_default_acl	права по умолчанию для различных типов объектов
pg_depend	зависимости между объектами базы данных
pg_description	описания или комментарии к объектам базы данных
pg_enum	определения меток и значений перечислений

Имя каталога	Предназначение
pg_event_trigger	событийные триггеры
pg_extension	установленные расширения
pg_foreign_data_wrapper	определения обёрток сторонних данных
pg_foreign_server	определения сторонних серверов
pg_foreign_table	дополнительные свойства сторонних таблиц
pg_index	дополнительные свойства индексов
pg_inherits	иерархия наследования таблиц
pg_init_privs	начальные права для объектов
pg_language	языки для написания функций
pg_largeobject	страницы данных для больших объектов
pg_largeobject_metadata	метаданные для больших объектов
pg_namespace	схемы
pg_opclass	классы операторов методов доступа
pg_operator	операторы
pg_opfamily	семейства операторов методов доступа
pg_partitioned_table	информация о ключах разбиения таблиц
pg_pltemplate	данные шаблонов для процедурных языков
pg_policy	политики защиты строк
pg_proc	функции и процедуры
pg_publication	публикации для логической репликации
pg_publication_rel	сопоставление отношений с публикациями
pg_range	информация о типах диапазонов
pg_replication_origin	зарегистрированные источники репликации
pg_rewrite	правила перезаписи запросов
pg_seclabel	метки безопасности для объектов базы данных
pg_sequence	информация о последовательностях
pg_shdepend	зависимости общих объектов
pg_shdescription	комментарии к общим объектам
pg_shseclabel	метки безопасности для общих объектов баз данных
pg_statistic	статистика планировщика
pg_statistic_ext	расширенная статистика планировщика
pg_subscription	подписки логической репликации
pg_subscription_rel	состояние отношений для подписок
pg_tablespace	табличные пространства в этом кластере баз данных
pg_transform	трансформации (тип данных для преобразований процедурных языков)
pg_trigger	триггеры
pg_ts_config	конфигурации текстового поиска

Имя каталога	Предназначение
pg_ts_config_map	сопоставления фрагментов в конфигурациях текстового поиска
pg_ts_dict	словари текстового поиска
pg_ts_parser	анализаторы текстового поиска
pg_ts_template	шаблоны текстового поиска
pg_type	типы данных
pg_user_mapping	сопоставления пользователей для сторонних серверов

49.2. pg_aggregate

В каталоге pg_aggregate хранится информация об агрегатных функциях. Агрегатная функция — это такая функция, которая работает с множеством значений (обычно, с содержимым одного столбца всех строк, удовлетворяющих условию запроса) и возвращает одно значение, вычисленное по этому множеству. Типичные агрегатные функции — sum, count и max. Все записи в pg_aggregate представляют собой дополнение записей в pg_proc. Запись в pg_proc определяет имя агрегатной функции, типы входных и выходных данных, а также другие свойства, подобные имеющимся у обычных функций.

Таблица 49.2. Столбцы pg_aggregate

Имя	Тип	Ссылки	Описание
aggfnoid	regproc	pg_proc .oid	OID агрегатной функции в pg_proc
aggkind	char		Тип агрегатной функции: n — обычная («normal»), o — сортирующая («ordered-set») или h — гипотезирующая («hypothetical-set»)
aggnumdirectargs	int2		Число непосредственных (не агрегируемых) аргументов для сортирующей или гипотезирующей агрегатной функции (переменный массив аргументов считается одним аргументом). Если равняется pronargs, агрегатная функция должна принимать переменный массив и этот массив описывает как агрегатные аргументы, так и окончательные непосредственные аргументы. Всегда равно нулю для

Имя	Тип	Ссылки	Описание
			обычных агрегатных функций.
aggtransfn	regproc	pg_proc .oid	Функция перехода
aggfinalfn	regproc	pg_proc .oid	Функция завершения (ноль, если её нет)
aggcombinefn	regproc	pg_proc .oid	Комбинирующая функция (ноль, если её нет)
aggserialfn	regproc	pg_proc .oid	Функция сериализации (ноль, если её нет)
aggdeserialfn	regproc	pg_proc .oid	Функция десериализации (ноль, если её нет)
aggmtransfn	regproc	pg_proc .oid	Функция прямого перехода для режима движущегося агрегата (ноль, если её нет)
aggminvtransfn	regproc	pg_proc .oid	Функция обратного перехода для режима движущегося агрегата (ноль, если её нет)
aggmfinalfn	regproc	pg_proc .oid	Функция завершения для режима движущегося агрегата (ноль, если её нет)
aggfinalextra	bool		Со значением True в aggfinalfn передаются дополнительные фиктивные аргументы
aggmfinalextra	bool		Со значением True в aggmfinalfn передаются дополнительные фиктивные аргументы
aggsortop	oid	pg_operator .oid	Связанный оператор сортировки (ноль, если его нет)
aggtranstype	oid	pg_type .oid	Тип данных внутреннего состояния (перехода) агрегатной функции
aggtransspace	int4		Приблизительный средний размер (в байтах) данных состояния перехода, либо ноль для выбора значения по умолчанию
aggmtranstype	oid	pg_type .oid	Тип данных внутреннего состояния (перехода) для

Имя	Тип	Ссылки	Описание
			агрегатной функции в режиме движущегося агрегата (ноль в случае отсутствия)
aggmtransspace	int4		Приблизительный средний размер (в байтах) данных состояния перехода для режима движущегося агрегата, либо ноль для выбора значения по умолчанию
agginitval	text		Начальное значение для состояния перехода. Это текстовое поле, содержащее значение в виде внешнего строкового представления. Если это поле содержит NULL, начальным значением состояния перехода будет NULL.
aggminitval	text		Начальное значение для состояния перехода в режиме движущегося агрегата. Это текстовое поле, содержащее значение в виде внешнего строкового представления. Если это поле содержит NULL, начальным значением состояния перехода будет NULL.

Новые агрегатные функции создаются командой CREATE AGGREGATE. За дополнительными сведениями о разработке агрегатных функций, значении функций перехода и т. д. обратитесь к Разделу 35.10.

49.3. pg_am

В каталоге pg_am хранится информация о методах доступа отношений. Каждая строка в нём описывает один метод доступа, поддерживаемый системой. В настоящее время методы доступа задаются только для индексов. Требования для реализации индексных методов доступа подробно рассматриваются в Главе 56.

Таблица 49.3. Столбцы pg_am

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
amname	name		Имя метода доступа

Имя	Тип	Ссылки	Описание
amhandler	regproc	pg_proc .oid	OID функции-обработчика, предоставляющей информацию о методе доступа
amtype	char		На данный момент это всегда i, что указывает, что это индексный метод доступа; в будущем могут появиться и другие значения

Примечание

До Postgres Pro 9.6, в pg_am было много дополнительных столбцов, представляющих свойства индексных методов доступа. Теперь эти данные непосредственно видны только на уровне кода C. Однако, чтобы SQL-запросы всё же могли проверять свойства индексных методов, была введена функция pg_index_column_has_property() и ряд связанных функций; см. Таблицу 9.63.

49.4. pg_amop

В каталоге pg_amop хранится информация об операторах, связанных с семействами операторов методов доступа. Каждая строка в нём описывает оператор, являющийся членом семейства операторов. Членом семейства может быть либо оператор *поиска*, либо оператор *упорядочивания*. Оператор может относиться к нескольким семействам, но он не может находиться в одном семействе в более чем одной позиции поиска или позиции упорядочивания. (Допустимо, хотя и маловероятно, что оператор будет использоваться и для поиска, и для упорядочивания.)

Таблица 49.4. Столбцы pg_amop

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
amopfamily	oid	pg_opfamily .oid	Семейство операторов, к которому относится эта запись
amoplefttype	oid	pg_type .oid	Тип данных левого операнда оператора
amoprighttype	oid	pg_type .oid	Тип данных правого операнда оператора
amopstrategy	int2		Номер стратегии оператора
amoppurpose	char		Предназначение оператора: s — поиск (search), o — упорядочивание (ordering)
amopopr	oid	pg_operator .oid	OID оператора

Имя	Тип	Ссылки	Описание
amopmethod	oid	pg_am .oid	Индексный метод доступа, для которого предназначено семейство операторов
amopsortfamily	oid	pg_opfamily .oid	Семейство операторов B-дерева, в соответствии с которым сортирует данный оператор, если это оператор упорядочивания; ноль, если это оператор поиска

Запись оператора «поиска» указывает, что по индексу этого семейства операторов можно производить поиск и найти все строки, удовлетворяющие условию WHERE *столбец_индекса оператор константа*. Очевидно, такой оператор должен возвращать тип boolean, а тип его левого операнда должен соответствовать типу данных столбца индекса.

Запись оператора «упорядочивания» указывает, что по индексу этого семейства операторов можно провести сканирование и получить строки в порядке, заданном предложением ORDER BY *столбец_индекса оператор константа*. Такой оператор может возвращать любой сортируемый тип данных, хотя тип левого операнда должен так же соответствовать типу данных столбца индекса. Точная семантика ORDER BY определяется столбцом amopsortfamily, который должна указывать на семейство операторов B-дерева для типа, возвращаемого оператором.

> **Примечание**
>
> В настоящее время предполагается, что порядком сортировки для упорядочивающего оператора будет порядок по умолчанию для соответствующего семейства операторов, то есть, ASC NULLS LAST. Когда-нибудь для большей гибкости могут быть добавлены дополнительные столбцы, позволяющие явно задавать параметры сортировки.

Поле amopmethod в записи оператора должно соответствовать полю opfmethod содержащего его семейства (amopmethod добавлен сюда намеренно, эта денормализация структуры каталога объясняется соображениями производительности). Также, поля amoplefttype и amoprighttype должны соответствовать полям oprleft и oprright в записи pg_operator, на которую ссылается данная.

49.5. pg_amproc

В каталоге pg_amproc хранится информация об опорных процедурах, связанных с семействами операторов методов доступа. Строки в нём описывают все опорные процедуры, принадлежащие семейству операторов.

Таблица 49.5. Столбцы pg_amproc

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
amprocfamily	oid	pg_opfamily .oid	Семейство операторов, к которому относится эта запись

Имя	Тип	Ссылки	Описание
amproclefttype	oid	pg_type .oid	Тип данных левого операнда связанного оператора
amprocrighttype	oid	pg_type .oid	Тип данных правого операнда связанного оператора
amprocnum	int2		Номер опорной процедуры
amproc	regproc	pg_proc .oid	OID процедуры

Обычно принято, что amproclefttype и amprocrighttype определяют типы левого и правого операнда оператора(ов), которые поддерживает конкретная опорная процедура. Для некоторых методов доступа они соответствуют типам входных данных самой опорной процедуры, для других — нет. Есть понятие «стандартных» опорных процедур для индекса; это такие процедуры, у которых amproclefttype и amprocrighttype равняются opcintype класса оператора индекса.

49.6. pg_attrdef

В каталоге pg_attrdef хранятся значения столбцов по умолчанию. Основная информация о столбцах хранится в pg_attribute (см. ниже). Данный же каталог содержит записи только для тех столбцов, для которых явно задаётся значение по умолчанию (при создании таблицы или добавлении столбца).

Таблица 49.6. Столбцы pg_attrdef

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
adrelid	oid	pg_class .oid	Таблица, к которой принадлежит столбец
adnum	int2	pg_attribute .attnum	Номер столбца
adbin	pg_node_tree		Внутреннее представление значения столбца по умолчанию
adsrc	text		Понятное человеку представление значения по умолчанию

Поле adsrc присутствует по исторически причинам, его не стоит использовать, так как в нём не отражаются внешние факторы, способные повлиять на представление значения по умолчанию. Для отображения значения по умолчанию лучше декомпилировать поле adbin (применив, например pg_get_expr).

49.7. pg_attribute

В каталоге pg_attribute хранится информация о столбцах таблицы. Для каждого столбца каждой таблицы в pg_attribute существует ровно одна строка. (Также в этом каталоге будут записи для индексов и на самом деле для всех объектов, присутствующих в pg_class.)

Термин «атрибут» равнозначен «столбцу» и употребляется по историческим причинам.

Таблица 49.7. Столбцы `pg_attribute`

Имя	Тип	Ссылки	Описание
attrelid	oid	pg_class .oid	Таблица, к которой принадлежит столбец
attname	name		Имя столбца
atttypid	oid	pg_type .oid	Тип данных этого столбца
attstattarget	int4		Столбец attstattarget управляет детализацией статистики, собираемой по этому столбцу командой ANALYZE. Нулевое значение указывает, что статистика не собирается. При отрицательном значении используется системное ограничение статистики по умолчанию. Точное значение положительных величин определяется типом данных. Для скалярных типов данных, attstattarget задаёт и целевое число собираемых «самых частых значений», и целевое число создаваемых групп гистограммы.
attlen	int2		Копия pg_type.typlen из записи типа столбца
attnum	int2		Порядковый номер столбца. Обычные столбцы нумеруются по возрастанию, начиная с 1. Системные столбцы, такие как oid, имеют (обычно) отрицательные номера.
attndims	int4		Число размерностей, если столбец имеет тип массива; ноль в противном случае. (В настоящее время число размерностей массива не контролируется, поэтому любое ненулевое значение по

Имя	Тип	Ссылки	Описание
			сути означает «это массив».)
attcacheoff	int4		Всегда -1 в постоянном хранилище, но когда запись загружается в память, в этом поле может кешироваться смещение атрибута в строке
atttypmod	int4		В поле atttypmod записывается дополнительное число, связанное с определённым типом данных, указываемое при создании таблицы (например, максимальный размер столбца varchar). Это значение передаётся функциям ввода и преобразования длины конкретного типа. Для типов, которым не нужен atttypmod, это обычно -1.
attbyval	bool		Копия pg_type.typbyval из записи типа столбца
attstorage	char		Обычно копия pg_type.typstorage из записи типа столбца. Для типов, поддерживающих TOAST, можно изменять это значение после создания столбца и таким образом управлять политикой хранения.
attalign	char		Копия pg_type.typalign из записи типа столбца
attnotnull	bool		Представляет ограничение NOT NULL.
atthasdef	bool		Столбец имеет значение по умолчанию, в этом случае в каталоге pg_attrdef будет соответствующая

Имя	Тип	Ссылки	Описание
			запись, определяющая это значение.
attidentity	char		Пустой символ ('') указывает, что это не столбец идентификации. Символ a указывает, что значение генерируется всегда, а d — что значение генерируется по умолчанию.
attisdropped	bool		Столбец был удалён и теперь не является рабочим. Удалённый столбец может по-прежнему физически присутствовать в таблице, но анализатор запросов его игнорирует, так что обратиться к нему из SQL нельзя.
attislocal	bool		Столбец определён локально в данном отношении. Заметьте, что столбец может быть определён локально и при этом наследоваться.
attinhcount	int4		Число прямых предков этого столбца. Столбец с ненулевым числом предков нельзя удалить или переименовать.
attcollation	oid	pg_collation .oid	Заданное для столбца правило сортировки, либо ноль, если тип столбца не сортируемый.
attacl	aclitem[]		Права доступа к столбцу, если они были заданы непосредственно для этого столбца
attoptions	text[]		Параметры уровня атрибута, в виде строк «ключ=значение»
attfdwoptions	text[]		Параметры уровня атрибута для обёрток сторонних данных, в виде строк «ключ=значение»

В записи удалённого столбца в pg_attribute поле atttypid сбрасывается в ноль, но attlen и другие поля, копируемые из pg_type, сохраняют актуальные значения. Это нужно, чтобы справиться с ситуацией, когда после удаления столбца удаляется и его тип данных, так что записи в pg_type больше не будет. В таких случаях для интерпретации содержимого строки таблицы могут использоваться attlen и другие поля.

49.8. pg_authid

В каталоге pg_authid содержится информация об идентификаторах для авторизации (ролях). Роль включает в себя концепции «пользователей» и «групп». Пользователь по существу представляет собой частный случай роли, с флагом rolcanlogin. Любая роль (с или без флага rolcanlogin) может включать другие роли в качестве членов; см. pg_auth_members.

Так как в этом каталоге содержатся пароли, он не должен быть открыт для всех. Для общего пользования предназначено представление pg_roles на базе pg_authid, в котором поле пароля очищено.

За подробной информацией о пользователях и управлении правами обратитесь к Главе 20.

Так как пользователи определяются глобально, каталог pg_authid разделяется всеми базами данных кластера; есть только единственная копия pg_authid в кластере, а не отдельные в каждой базе данных.

Таблица 49.8. Столбцы pg_authid

Имя	Тип	Описание
oid	oid	Идентификатор строки (скрытый атрибут; должен выбираться явно)
rolname	name	Имя роли
rolsuper	bool	Роли имеет права суперпользователя
rolinherit	bool	Роль автоматически наследует права ролей, в которые она включена
rolcreaterole	bool	Роль может создавать другие роли
rolcreatedb	bool	Роль может создавать базы данных
rolcanlogin	bool	Роль может подключаться к серверу. То есть эта роль может быть указана в качестве начального идентификатора авторизации сеанса.
rolreplication	bool	Роль является ролью репликации. Такие роли могут устанавливать соединения для репликации и создавать/удалять слоты репликации.
rolbypassrls	bool	Роль не подчиняется никаким политикам защиты на уровне строк; за подробностями обратитесь к Разделу 5.7.
rolconnlimit	int4	Для ролей, которые могут подключаться к серверу, это

Имя	Тип	Описание
		значение задаёт максимально разрешённое для этой роли число одновременных подключений. При значении -1 ограничения нет.
rolpassword	text	Пароль (возможно зашифрованный); NULL, если он не задан. Его формат зависит от используемого вида шифрования.
rolvaliduntil	timestamptz	Срок действия пароля (используется только при аутентификации по паролю); NULL, если срок действия не ограничен

Если пароль зашифрован MD5, значение в rolpassword начинается со строки md5, за которой идёт 32-символьный шестнадцатеричный хеш MD5. Этот хеш вычисляется для пароля пользователя с добавленным за ним его именем. Например, если у пользователя joe пароль xyzzy, Postgres Pro сохранит в этом поле md5-хеш строки xyzzyjoe.

Если пароль зашифрован по алгоритму SCRAM-SHA-256, он имеет формат:

SCRAM-SHA-256$*<число итераций>*:*<соль>*$*<СохранённыйКлюч>*:*<КлючСервера>*

Здесь *соль*, *СохранённыйКлюч* и *КлючСервера* кодируются в формате Base64. Этот формат соответствует стандарту RFC 5803.

Пароль, не удовлетворяющий ни одному из этих форматов, считается незашифрованным.

49.9. pg_auth_members

Каталог pg_auth_members представляет отношения членства между ролями. Допускается любая не зацикленная иерархия отношений.

Так как пользователи определяются глобально, pg_auth_members разделяется всеми базами данных кластера; есть только единственная копия pg_auth_members в кластере, а не отдельные в каждой базе данных.

Таблица 49.9. Столбцы pg_auth_members

Имя	Тип	Ссылки	Описание
roleid	oid	pg_authid .oid	Идентификатор роли, включающей другую
member	oid	pg_authid .oid	Идентификатор роли, включаемой в роль roleid
grantor	oid	pg_authid .oid	Идентификатор роли, разрешившей членство
admin_option	bool		True, если member может разрешать членство в roleid другим ролям

49.10. pg_cast

В каталоге pg_cast хранятся пути приведения типов (как встроенных, так и пользовательских).

Следует заметить, что pg_cast представляет не каждое приведение, которое может выполнять система, а только те, которые нельзя вывести по некоторым общим правилам. Например, приведение типа домена к его базовому типу не представляется явно в pg_cast. Ещё одно важное исключение составляют «автоматические приведения ввода/вывода», которые выполняются с применением собственных функций ввода/вывода типа данных, преобразующих тип в text или другие строковые типы — они также не представляются явно в pg_cast.

Таблица 49.10. Столбцы pg_cast

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
castsource	oid	pg_type .oid	OID исходного типа данных
casttarget	oid	pg_type .oid	OID целевого типа данных
castfunc	oid	pg_proc .oid	OID функции, выполняющей приведение, или ноль, если для данного метода приведения не требуется функция.
castcontext	char		Определяет, в каких контекстах может выполняться приведение. Символ e означает только явное приведение (explicit), с применением синтаксиса CAST или ::. Символ a означает неявное присваивание (assignment) целевому столбцу, а также явное приведение. Символ i разрешает неявное приведение (implicit), а также все остальные варианты.
castmethod	char		Показывает, как выполняется приведение. Символ f означает, что используется функция, указанная в поле castfunc. Символ i означает, что используются функции ввода/вывода. Символ b означают, что типы являются двоично-сводимыми, так что преобразование не требуется.

Функции приведения, перечисленные в pg_cast, должны всегда принимать исходный тип приведения в качестве типа первого аргумента и возвращать результат, имеющий целевой тип. Функция приведения может иметь до трёх аргументов. Второй аргумент, если он присутствует, должен иметь тип integer; в нём передаётся модификатор типа, связанный с целевым типом, либо -1, если такого модификатора нет. Третий аргумент, если он присутствует, должен иметь тип boolean; в нём передаётся true, если приведение выполняется явно, и false в противном случае.

Вполне возможно создать запись в pg_cast, в которой исходный тип будет совпадать с целевым, если связанная функция приведения принимает больше одного аргумента. Такие записи представляют «функции преобразования длины», которые приводят значения некоторого типа в соответствие с заданным значением модификатора.

Когда в записи pg_cast исходный и целевой типы приведения различаются и функция принимает более одного аргумента, эта запись представляет преобразование типа из одного в другой и сведение к нужной длине за один шаг. Если же такой записи не находится, приведение к типу с определённым модификатором выполняется в два этапа, сначала выполняется преобразование типа, а затем применяется модификатор типа.

49.11. pg_class

В каталоге pg_class описываются таблицы и практически всё, что имеет столбцы или каким-то образом подобно таблице. Сюда входят индексы (но смотрите также pg_index), последовательности (но смотрите также pg_sequence), представления, материализованные представления, составные типы и таблицы TOAST; см. relkind. Далее, подразумевая все эти типы объектов, мы будем говорить об «отношениях». Не все столбцы здесь имеют смысл для всех типов отношений.

Таблица 49.11. Столбцы pg_class

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
relname	name		Имя таблицы, индекса, представления и т. п.
relnamespace	oid	pg_namespace .oid	OID пространства имён, содержащего это отношение
reltype	oid	pg_type .oid	OID типа данных, соответствующего типу строки этой таблицы, если таковой есть (ноль для индексов, так как они не имеют записи в pg_type)
reloftype	oid	pg_type .oid	Для типизированных таблиц, OID нижележащего составного типа, или ноль для всех других отношений
relowner	oid	pg_authid .oid	Владелец отношения
relam	oid	pg_am .oid	Если это индекс, применяемый метод

Имя	Тип	Ссылки	Описание
			доступа (B-дерево, хеш и т. д.)
relfilenode	oid		Имя файла на диске с этим отношением; ноль означает, что это «отображённое» представление, имя файла для которого определяется состоянием на нижнем уровне
reltablespace	oid	pg_tablespace .oid	Табличное пространство, в котором хранится это отношение. Если ноль, подразумевается пространство базы данных по умолчанию. (Не имеет значения, если с отношением не связан файл на диске.)
relpages	int4		Размер представления этой таблицы на диске (в страницах размера BLCKSZ). Это лишь примерная оценка, используемая планировщиком. Она обновляется командами VACUUM, ANALYZE и несколькими командами DDL, например, CREATE INDEX.
reltuples	float4		Число строк в таблице. Это лишь примерная оценка, используемая планировщиком. Она обновляется командами VACUUM, ANALYZE и несколькими командами DDL, например, CREATE INDEX.
relallvisible	int4		Число страниц, помеченных как «полностью видимые» в карте видимости таблицы. Это лишь примерная оценка, используемая планировщиком. Она обновляется командами VACUUM,

Имя	Тип	Ссылки	Описание
			ANALYZE и несколькими командами DDL, например, CREATE INDEX.
reltoastrelid	oid	pg_class .oid	OID таблицы TOAST, связанной с данной таблицей, или 0, если таковой нет. В таблицу TOAST, как во вторичную, «выносятся» большие атрибуты.
relhasindex	bool		True, если это таблица и она имеет (или недавно имела) индексы
relisshared	bool		True, если эта таблица разделяется всеми базами данных в кластере. Разделяемыми являются только некоторые системные каталоги (как например, pg_database).
relpersistence	char		p = постоянная таблица (permanent), u = нежурналируемая таблица (unlogged), t = временная таблица (temporary)
relkind	char		r = обычная таблица (Relation), i = индекс (Index), S = последовательность (Sequence), t = таблица TOAST, v = представление (View), m = материализованное представление (Materialized view), c = составной тип (Composite type), f = сторонняя таблица (Foreign table), p = секционированная таблица (Partitioned table)
relnatts	int2		Число пользовательских столбцов в отношении (системные столбцы не

Имя	Тип	Ссылки	Описание
			считаются). Столько же соответствующих строк должно быть в pg_attribute . См. также pg_attribute.attnum .
relchecks	int2		Число ограничений CHECK в таблице; см. каталог pg_constraint
relhasoids	bool		True, если для каждой строки отношения генерируется OID
relhaspkey	bool		True, если в таблице имеется (или имелся) первичный ключ
relhasrules	bool		True, если для таблицы определены (или были определены) правила; см. каталог pg_rewrite
relhastriggers	bool		True, если для таблицы определены (или были определены) триггеры; см. каталог pg_trigger
relhassubclass	bool		True, если у таблицы есть (или были) потомки в иерархии наследования
relrowsecurity	bool		True, если для таблицы включена защита на уровне строк; см. каталог pg_policy
relforcerowsecurity	bool		True, если защита на уровне строк (когда она включена) также применяется к владельцу таблицы; см. каталог pg_policy
relispopulated	bool		True, если отношение наполнено данными (это истинно для всех отношений, кроме некоторых материализованных представлений)
relreplident	char		Столбцы, формирующие «идентификатор реплики» для строк: d = по умолчанию (первичный ключ, если

Имя	Тип	Ссылки	Описание
			есть), `n` = никакие (nothing), `f` = все столбцы, `i` = индекс со свойством `indisreplident` (если ранее использованный индекс удалён, действует так же, как `n`)
`relispartition`	`bool`		True, если таблица является секцией
`relfrozenxid`	`xid`		Идентификаторы транзакций, предшествующие данному, в этой таблице заменены постоянным («замороженным») идентификатором транзакции. Это нужно для определения, когда требуется очищать таблицу для предотвращения зацикливания идентификаторов или для сокращения объёма `pg_xact` . Если это отношение — не таблица, значение равно нулю (`InvalidTransactionId`).
`relminmxid`	`xid`		Идентификаторы мультитранзакций, предшествующие данному, в этой таблице заменены другим идентификатором транзакции. Это нужно для определения, когда требуется очищать таблицу для предотвращения зацикливания идентификаторов мультитранзакций или для сокращения объёма `pg_multixact` . Если это отношение — не таблица, значение равно нулю (`InvalidMultiXactId`).

Имя	Тип	Ссылки	Описание
relacl	aclitem[]		Права доступа; за подробностями обратитесь к описанию GRANT и REVOKE
reloptions	text[]		Специальные параметры для методов доступа, в виде строк «ключ=значение»
relpartbound	pg_node_tree		Если таблица является секцией (см. relispartition), внутреннее представление границ секции

Некоторые логические флаги в pg_class поддерживаются не строго: гарантируется, что они будут установлены при переходе в определённое состояние, но они могут не сбрасываться немедленно, когда условия поменяются. Например, relhasindex устанавливается командой CREATE INDEX, но никогда не сбрасывается командой DROP INDEX. Вместо этого, флаг relhasindex сбрасывается командой VACUUM, если она находит, что в таблице нет индексов. Такая организация позволяет избежать состояния гонки и способствует параллельному использованию.

49.12. pg_collation

В каталоге pg_collation описываются доступные правила сортировки, которые по сути представляют собой сопоставления идентификаторов SQL с категориями локалей операционной системы. За дополнительными сведениями обратитесь к Разделу 22.2.

Таблица 49.12. Столбцы pg_collation

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
collname	name		Имя правила сортировки (уникальное для пространства имён и кодировки)
collnamespace	oid	pg_namespace .oid	OID пространства имён, содержащего это правило сортировки
collowner	oid	pg_authid .oid	Владелец правила сортировки
collprovider	char		Провайдер правила сортировки: d = установленный в базе по умолчанию, c = libc, i = icu
collencoding	int4		Кодировка, для которой применимо это правило, или -1, если

Имя	Тип	Ссылки	Описание
			оно работает с любой кодировкой
collcollate	name		LC_COLLATE для данного объекта
collctype	name		LC_CTYPE для данного объекта
collversion	text		Определяемая провайдером версия правила сортировки. Она записывается при создании правила сортировки и проверяется при использовании для обнаружения изменений в его определении, чреватых повреждением данных.

Заметьте, что уникальный ключ в этом каталоге определён как (collname, collencoding, collnamespace), а не просто как (collname, collnamespace). Вообще Postgres Pro игнорирует все правила сортировки, для которых collencoding не равняется кодировке текущей базы данных или -1, а создание новых записей с тем же именем, которое уже имеет запись с collencoding = -1, запрещено. Таким образом, достаточно использовать полное имя SQL (*схема*.*имя*) для указания правила сортировки, несмотря на то, что оно может быть неуникальным согласно определению каталога. Такая организация каталога объясняется тем, что программа initdb наполняет его в момент инициализации кластера записями для всех локалей, обнаруженных в системе, так что она должна иметь возможность сохранить записи для всех кодировок, которые могут вообще когда-либо применяться в кластере.

В базе данных template0 может быть полезно создать правила сортировки, кодировки которых не соответствуют кодировке этой базы, но которые могут оказаться у баз данных, скопированных впоследствии из template0. В настоящее время это придётся проделать вручную.

49.13. pg_constraint

В каталоге pg_constraint хранятся ограничения-проверки, ограничения-исключения, а также ограничения первичного ключа, уникальности и внешних ключей, определённые для таблиц. (Ограничения столбцов описываются как и все остальные. Любое ограничение столбца равнозначно некоторому ограничению таблицы.) Ограничения на NULL представляются не здесь, а в каталоге pg_attribute.

Для пользовательских триггеров ограничений (создаваемых командой CREATE CONSTRAINT TRIGGER) в этой таблице также создаётся запись.

Здесь также хранятся ограничения доменов.

Таблица 49.13. Столбцы pg_constraint

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)

Имя	Тип	Ссылки	Описание
conname	name		Имя ограничения (не обязательно уникальное!)
connamespace	oid	pg_namespace .oid	OID пространства имён, содержащего это ограничение
contype	char		c = ограничение-проверка (check), f = внешний ключ (foreign key), p = первичный ключ (primary key), u = ограничение уникальности (unique), t = триггер ограничения (trigger), x = ограничение-исключение (exclusion)
condeferrable	bool		Является ли ограничение откладываемым?
condeferred	bool		Является ли ограничение отложенным по умолчанию?
convalidated	bool		Было ли ограничение проверено? В настоящее время значение false возможно только для внешних ключей и ограничений CHECK
conrelid	oid	pg_class .oid	Таблица, для которой установлено это ограничение; 0, если это не ограничение таблицы
contypid	oid	pg_type .oid	Домен, к которому относится это ограничение; 0, если это не ограничение домена
conindid	oid	pg_class .oid	Индекс, поддерживающий это ограничение, если это ограничение уникальности, первичного или внешнего ключа, либо ограничение-исключение; в противном случае — 0
confrelid	oid	pg_class .oid	Если это внешний ключ, таблица, на

Имя	Тип	Ссылки	Описание
			которую он ссылается; иначе 0
confupdtype	char		Код действия при изменении внешнего ключа: a = нет действия, r = ограничить (restrict), c = каскадное действие (cascade), n = присвоить NULL, d = поведение по умолчанию
confdeltype	char		Код действия при удалении внешнего ключа: a = нет действия, r = ограничить (restrict), c = каскадное действие (cascade), n = присвоить NULL, d = поведение по умолчанию
confmatchtype	char		Тип сопоставления внешнего ключа: f = полное (full), p = частичное (partial), s = простое (simple)
conislocal	bool		Ограничение определено локально в данном отношении. Заметьте, что ограничение может быть определено локально и при этом наследоваться.
coninhcount	int4		Число прямых предков этого ограничения. Ограничение с ненулевым числом предков нельзя удалить или переименовать.
connoinherit	bool		Ограничение определено локально для данного отношения и является ненаследуемым.
conkey	int2[]	pg_attribute .attnum	Для ограничений таблицы (включая внешние ключи, но не триггеры ограничений), определяет список столбцов, образующих ограничение

Имя	Тип	Ссылки	Описание
confkey	int2[]	pg_attribute .attnum	Для внешнего ключа определяет список столбцов, на которые он ссылается
conpfeqop	oid[]	pg_operator .oid	Для внешнего ключа — список операторов равенства для сравнений PK = FK
conppeqop	oid[]	pg_operator .oid	Для внешнего ключа — список операторов равенства для сравнений PK = PK
conffeqop	oid[]	pg_operator .oid	Для внешнего ключа — список операторов равенства для сравнений FK = FK
conexclop	oid[]	pg_operator .oid	Для ограничения-исключения — список операторов исключения по столбцам
conbin	pg_node_tree		Для ограничения-проверки — внутреннее представление выражения
consrc	text		Для ограничения-проверки — понятное человеку представление выражения

В случае с ограничением-исключением значение conkey полезно только для элементов ограничений, представляющих простые ссылки на столбцы. Для других случаев в conkey задаётся ноль, и чтобы получить выражение, определяющее ограничение, надо обратиться к соответствующему индексу. (Таким образом, поле conkey имеет то же содержимое, что и pg_index.indkey для индекса.)

Примечание

Поле consrc не меняется при изменении задействованных в выражении объектов; например, в нём не отслеживается переименование столбцов. Поэтому лучше не полагаться на него, а воспользоваться функцией pg_get_constraintdef(), чтобы получить определение ограничения-проверки.

Примечание

Поле pg_class.relchecks должно согласовываться с числом ограничений-проверок, описанных в данной таблице для каждого отношения.

49.14. pg_conversion

В каталоге pg_conversion описываются процедуры преобразования кодировки. За дополнительными сведениями обратитесь к CREATE CONVERSION.

Таблица 49.14. Столбцы pg_conversion

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
conname	name		Имя перекодировки (уникальное в пространстве имён)
connamespace	oid	pg_namespace .oid	OID пространства имён, содержащего эту перекодировку
conowner	oid	pg_authid .oid	Владелец перекодировки
conforencoding	int4		Идентификатор исходной кодировки
contoencoding	int4		Идентификатор целевой кодировки
conproc	regproc	pg_proc .oid	Процедура преобразования
condefault	bool		True, если это перекодировка по умолчанию

49.15. pg_database

В каталоге pg_database хранится информация о доступных базах данных. Базы данных создаются командой CREATE DATABASE. Подробнее о предназначении некоторых свойств баз можно узнать в Главе 21.

В отличие от большинства системных каталогов, pg_database разделяется всеми базами данных кластера: есть только один экземпляр pg_database в кластере, а не отдельные в каждой базе данных.

Таблица 49.15. Столбцы pg_database

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
datname	name		Имя базы данных
datdba	oid	pg_authid .oid	Владелец базы данных, обычно пользователь, создавший её
encoding	int4		Кодировка символов для этой базы данных (pg_encoding_to_char() может преобразовать этот номер в имя кодировки)

Имя	Тип	Ссылки	Описание
datcollate	name		LC_COLLATE для этой базы данных
datctype	name		LC_CTYPE для этой базы данных
datistemplate	bool		Если true, базу данных сможет клонировать любой пользователь с правами CREATEDB; в противном случае клонировать эту базу смогут только суперпользователи и её владелец.
datallowconn	bool		Если false, никто не сможет подключаться к этой базе данных. Это позволяет защитить базу данных template0 от модификаций.
datconnlimit	int4		Задаёт максимально допустимое число одновременных подключений к этой базе данных. С -1 ограничения нет.
datlastsysoid	oid		Последний системный OID в базе данных; в частности, полезен для pg_dump
datfrozenxid	xid		Все идентификаторы транзакций, предшествующие данному, в этой базе данных заменены постоянным («замороженным») идентификатором транзакции. Это нужно для определения, когда требуется очищать базу данных для предотвращения зацикливания идентификаторов или для сокращения объёма pg_xact . Это значение вычисляется как минимум значений pg_class .relfrozenxid для всех таблиц.
datminmxid	xid		Идентификаторы мультитранзакций, предшествующие

Имя	Тип	Ссылки	Описание
			данному, в этой базе данных заменены другим идентификатором транзакции. Это нужно для определения, когда требуется очищать базу данных для предотвращения зацикливания идентификаторов мультитранзакций или для сокращения объёма pg_multixact . Это значение вычисляется как минимум значений pg_class .relminmxid для всех таблиц.
dattablespace	oid	pg_tablespace .oid	Табличное пространство по умолчанию для данной базы данных. Если таблица базы находится в этом пространстве, для неё значение pg_class .reltablespace будет нулевым; в частности, в нём окажутся все частные системные каталоги этой базы.
datacl	aclitem[]		Права доступа; за подробностями обратитесь к описанию GRANT и REVOKE

49.16. pg_db_role_setting

В каталоге pg_db_role_setting записываются значения по умолчанию, присваиваемые переменным конфигурации во время выполнения, для различных комбинаций ролей и баз данных.

В отличие от большинства системных каталогов, pg_db_role_setting разделяется всеми базами данных кластера: есть только один экземпляр pg_db_role_setting в кластере, а не отдельные в каждой базе данных.

Таблица 49.16. Столбцы pg_db_role_setting

Имя	Тип	Ссылки	Описание
setdatabase	oid	pg_database .oid	OID базы данных, к которой относится это присваивание переменных, или ноль, если оно не связано с базой данных

Имя	Тип	Ссылки	Описание
setrole	oid	pg_authid .oid	OID роли, к которой применимо это присваивание переменных, или ноль, если оно не связано с ролью
setconfig	text[]		Значения по умолчанию для переменных конфигурации времени выполнения

49.17. pg_default_acl

В каталоге pg_default_acl хранятся определения прав, изначально присваиваемые создаваемым объектам.

Таблица 49.17. Столбцы pg_default_acl

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
defaclrole	oid	pg_authid .oid	OID роли, связанной с этой записью
defaclnamespace	oid	pg_namespace .oid	OID пространства имён, связанного с этой записью, или 0, если запись глобальная
defaclobjtype	char		Тип объекта, права для которого определяет эта запись: r = отношение (relation) (таблица, представление), S = последовательность (sequence), f = функция (function), T = тип (type), n = схема (schema)
defaclacl	aclitem[]		Права доступа, назначаемые объекту данного типа при создании

В каталоге pg_default_acl описываются начальные права доступа, которые будут связаны с объектом, принадлежащим заданному пользователю. В настоящее время есть два типа записей: «глобальные», с defaclnamespace равным нулю, и «внутрисхемные», относящиеся к конкретной схеме. Если присутствует глобальная запись, она *переопределяет* обычный жёстко фиксированный набор прав для данного типа объектов. Если присутствует внутрисхемная запись, она представляет набор прав, *добавляемый* к набору прав, определённых глобально или жёстко заданных по умолчанию.

Заметьте, что когда поле ACL в другом каталоге содержит NULL, под этим подразумеваются жёстко заданные права по умолчанию для этого объекта, а *не* те, которые могут быть в pg_default_acl в данный момент. Права в pg_default_acl учитываются только при создании объекта.

49.18. pg_depend

В каталоге pg_depend записываются отношения зависимости между объектами базы данных. Благодаря этой информации, команды DROP могут найти, какие объекты должны удаляться при использовании DROP CASCADE, или когда нужно запрещать удаление при DROP RESTRICT.

Также смотрите описание каталога pg_shdepend, который играет подобную роль в отношении совместно используемых объектов в кластере баз данных.

Таблица 49.18. Столбцы pg_depend

Имя	Тип	Ссылки	Описание
classid	oid	pg_class .oid	OID системного каталога, в котором находится зависимый объект
objid	oid	любой столбец OID	OID определённого зависимого объекта
objsubid	int4		Для столбца таблицы это номер столбца (objid и classid указывают на саму таблицу). Для всех других типов объектов это поле содержит ноль.
refclassid	oid	pg_class .oid	OID системного каталога, в котором находится вышестоящий объект
refobjid	oid	любой столбец OID	OID определённого вышестоящего объекта
refobjsubid	int4		Для столбца таблицы это номер столбца (refobjid и refclassid указывают на саму таблицу). Для всех других типов объектов это поле содержит ноль.
deptype	char		Код, определяющий конкретную семантику данного отношения зависимости; см. текст

Во всех случаях, запись в pg_depend показывает, что вышестоящий объект нельзя удалить, не удаляя подчинённый объект. Однако есть несколько подвидов зависимости, задаваемых в поле deptype:

DEPENDENCY_NORMAL (n)

Обычное отношение между отдельно создаваемыми объектами. Подчинённый объект можно удалить, не затрагивая вышестоящий объект. Вышестоящий объект можно удалить только с

указанием CASCADE, при этом будет удалён и подчинённый объект. Например, столбец таблицы находится в обычной зависимости от своего типа данных.

DEPENDENCY_AUTO (a)

Подчинённый объект может быть удалён отдельно от вышестоящего и должен быть удалён автоматически (вне зависимости от указаний RESTRICT и CASCADE), если удаляется вышестоящий объект. Например, именованное ограничение для таблицы находится в автоматической зависимости от таблицы, так что оно исчезнет при удалении таблицы.

DEPENDENCY_INTERNAL (i)

Подчинённый объект был создан в процессе создания вышестоящего и на самом деле является только частью его внутренней реализации. Для такого объекта будет запрещена команда DROP (мы подскажем пользователю, что вместо этого надо выполнить DROP вышестоящий объект). Действие DROP для вышестоящего объекта будет распространено и на этот подчинённый объект, вне зависимости от присутствия указания CASCADE. Например, триггер, созданный для обеспечения ограничения внешнего ключа, становится внутренне зависимым от записи ограничения в pg_constraint.

DEPENDENCY_EXTENSION (e)

Подчинённый объект входит в состав *расширения*, которое является вышестоящим объектом (см. pg_extension). Удалить подчинённый объект можно, только выполнив команду DROP EXTENSION для вышестоящего объекта. Функционально этот тип зависимости действует так же, как и внутренняя зависимость, но он выделен для наглядности и упрощения pg_dump.

DEPENDENCY_AUTO_EXTENSION (x)

Подчинённый объект не входит в состав расширения, которое является вышестоящим объектом (и поэтому не должен игнорироваться утилитой pg_dump), но не может функционировать без него, и поэтому должен удаляться при удалении расширения. При этом подчинённый объект может быть удалён и сам по себе.

DEPENDENCY_PIN (p)

Зависимый объект отсутствует; этот тип записи показывает, что система сама зависит от вышестоящего объекта, так что этот объект нельзя удалять ни при каких условиях. Записи этого типа создаются только командой initdb. Поля зависимого объекта в такой записи содержат нули.

В будущем могут появиться и другие подвиды зависимости.

49.19. pg_description

В каталоге pg_description хранятся дополнительные описания (комментарии) для объектов баз данных. Описания можно задавать с помощью команды COMMENT и просматривать в psql, используя команды \d. В начальном содержимом pg_description представлены описания многих встроенных системных объектов.

Также смотрите каталог pg_shdescription, который играет подобную роль в отношении совместно используемых объектов в кластере баз данных.

Таблица 49.19. Столбцы pg_description

Имя	Тип	Ссылки	Описание
objoid	oid	любой столбец OID	OID объекта, к которому относится это описание
classoid	oid	pg_class .oid	OID системного каталога, к которому относится этот объект

Имя	Тип	Ссылки	Описание
objsubid	int4		Для комментария к столбцу таблицы это номер столбца (objoid и classoid указывают на саму таблицу). Для всех других типов объектов это поле содержит ноль.
description	text		Произвольный текст, служащий описанием данного объекта

49.20. pg_enum

В каталоге pg_enum содержатся записи, определяющие значения и метки для всех типов-перечислений. Внутренним представлением значения перечисления на самом деле является OID соответствующей строки в pg_enum.

Таблица 49.20. Столбцы pg_enum

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
enumtypid	oid	pg_type .oid	OID типа в pg_type , к которому относится данное значение перечисления
enumsortorder	float4		Порядок сортировки этого значения внутри перечисления
enumlabel	name		Текстовая метка данного значения перечисления

Идентификаторы OID в строках pg_enum подчиняются особому правилу: чётные OID гарантированно упорядочиваются по порядку сортировки их типа перечисления. То есть, если к одному перечислению относятся два чётных OID, меньшему OID должно соответствовать меньшее значение enumsortorder. Нечётные значения OID могут быть не связаны с этим порядком сортировки. Это правило позволяет во многих случаях сравнивать значения перечислений, не обращаясь к каталогам. Процедуры, создающие и изменяющие перечисления, пытаются присваивать значениям перечислений чётные OID, если это возможно.

Когда создаётся тип перечисления, его членам назначаются позиции по порядку сортировки 1..*n*. Но у членов, добавляемых позже, могут оказаться отрицательные или дробные значения enumsortorder. Единственное, что требуется — чтобы эти значения были правильно упорядочены и уникальны в рамках перечисления.

49.21. pg_event_trigger

В каталоге pg_event_trigger хранится информация о событийных триггерах. За дополнительными сведениями обратитесь к Главе 37.

Таблица 49.21. Столбцы pg_event_trigger

Имя	Тип	Ссылки	Описание
evtname	name		Имя триггера (должно быть уникальным)
evtevent	name		Указывает, для какого события срабатывает данный триггер
evtowner	oid	pg_authid .oid	Владелец событийного триггера
evtfoid	oid	pg_proc .oid	Вызываемая функция
evtenabled	char		Устанавливает, в каких режимах session_replication_role срабатывает событийный триггер: O = триггер срабатывает в режимах «origin» (источник) и «local» (локально), D = триггер отключён, R = триггер срабатывает в режиме «replica» (реплика), A = триггер срабатывает всегда.
evttags	text[]		Теги команд, для которых будет срабатывать триггер. Если NULL, срабатывание триггера не будет ограничено в зависимости от тега команды.

49.22. pg_extension

В каталоге pg_extension хранится информация об установленных расширениях. Подробнее о расширениях можно узнать в Разделе 35.15.

Таблица 49.22. Столбцы pg_extension

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
extname	name		Имя расширения
extowner	oid	pg_authid .oid	Владелец расширения
extnamespace	oid	pg_namespace .oid	Схема, содержащая экспортируемые расширением объекты
extrelocatable	bool		True, если расширение можно переместить в другую схему

Имя	Тип	Ссылки	Описание
extversion	text		Имя версии расширения
extconfig	oid[]	pg_class .oid	Массив с идентификаторами regclass, указывающими на таблицы конфигурации расширения, либо NULL, если таких таблиц нет
extcondition	text[]		Массив с условиями фильтра WHERE для таблиц конфигурации расширения, либо NULL, если таких условий нет

Заметьте, что в отличие от большинства каталогов со столбцом «namespace», здесь extnamespace не подразумевает, что расширение принадлежит данной схеме. Имена расширений никогда не дополняются схемой. Вместо этого, extnamespace показывает, что в этой схеме содержатся все или большинство объектов расширения. Если extrelocatable имеет значение true, эта схема должна фактически содержать все относящиеся к схеме объекты, составляющие это расширение.

49.23. pg_foreign_data_wrapper

В каталоге pg_foreign_data_wrapper хранятся определения обёрток сторонних данных. Обёрткой сторонних данных называется механизм, через который становятся доступны внешние данные, расположенные на сторонних серверах.

Таблица 49.23. Столбцы pg_foreign_data_wrapper

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
fdwname	name		Имя обёртки сторонних данных
fdwowner	oid	pg_authid .oid	Владелец обёртки сторонних данных
fdwhandler	oid	pg_proc .oid	Указывает на функцию-обработчик, отвечающую за выдачу набора исполняемых процедур для обёртки сторонних данных. Ноль говорит об отсутствии такого обработчика
fdwvalidator	oid	pg_proc .oid	Указывает на функцию проверки, которая отвечает за проверку правильности параметров, передаваемых обёртке сторонних данных, а

Имя	Тип	Ссылки	Описание
			также параметров для сторонних серверов и сопоставлений пользователей, задаваемых через эту обёртку. Ноль говорит об отсутствии такой функции
fdwacl	aclitem[]		Права доступа; за подробностями обратитесь к описанию GRANT и REVOKE
fdwoptions	text[]		Специальные параметры обёртки сторонних данных, в виде строк «ключ=значение»

49.24. pg_foreign_server

В каталоге pg_foreign_server хранятся определения сторонних серверов. Запись стороннего сервера описывает источник внешних данных, например удалённый сервер. Обращение к сторонним серверам происходит через обёртки сторонних данных.

Таблица 49.24. Столбцы pg_foreign_server

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
srvname	name		Имя стороннего сервера
srvowner	oid	pg_authid .oid	Владелец стороннего сервера
srvfdw	oid	pg_foreign_data_wrapper .oid	OID обёртки сторонних данных для этого стороннего сервера
srvtype	text		Тип сервера (необязателен)
srvversion	text		Версия сервера (необязательна)
srvacl	aclitem[]		Права доступа; за подробностями обратитесь к описанию GRANT и REVOKE
srvoptions	text[]		Специальные параметры стороннего сервера, в виде строк «ключ=значение»

49.25. pg_foreign_table

В каталоге pg_foreign_table содержится дополнительная информация о сторонних таблицах. Прежде всего сторонняя таблица представляется записью в pg_class, как и обычная. Запись в pg_foreign_table для неё содержит свойства, применимые только к сторонним таблицам, но не к каким-либо другим типам отношений.

Таблица 49.25. Столбцы pg_foreign_table

Имя	Тип	Ссылки	Описание
ftrelid	oid	pg_class .oid	OID записи в pg_class для этой сторонней таблицы
ftserver	oid	pg_foreign_server .oid	OID стороннего сервера для этой сторонней таблицы
ftoptions	text[]		Параметры сторонней таблицы, в виде строк «ключ=значение»

49.26. pg_index

В каталоге pg_index содержится часть информации об индексах. Остальная информация в основном находится в pg_class.

Таблица 49.26. Столбцы pg_index

Имя	Тип	Ссылки	Описание
indexrelid	oid	pg_class .oid	OID записи в pg_class для этого индекса
indrelid	oid	pg_class .oid	OID записи в pg_class для таблицы, к которой относится этот индекс
indnatts	int2		Число столбцов в индексе (повторяет значение pg_class.relnatts)
indnkeyatts	int2		Количество ключевых столбцов в индексе. «Ключевые столбцы» это обычные образующие индекс столбцы, в отличие от «включённых» столбцов.
indisunique	bool		Если true, это уникальный индекс
indisprimary	bool		Если true, этот индекс представляет первичный ключ таблицы (в этом случае и в поле indisunique должно быть значение true)
indisexclusion	bool		Если true, этот индекс поддерживает

Имя	Тип	Ссылки	Описание
			ограничение-исключение
indimmediate	bool		Если true, проверка уникальности осуществляется непосредственно при добавлении данных (неприменимо, если значение indisunique не true)
indisclustered	bool		Если true, таблица в последний раз кластеризовалась по этому индексу
indisvalid	bool		Если true, индекс можно применять в запросах. Значение false означает, что индекс, возможно, неполный: он будет тем не менее изменяться командами INSERT/UPDATE, но безопасно применять его в запросах нельзя. Если он уникальный, свойство уникальности также не гарантируется.
indcheckxmin	bool		Если true, запросы не должны использовать этот индекс, пока поле xmin данной записи в pg_index не окажется ниже их горизонта событий TransactionXmin, так как таблица может содержать оборванные цепочки HOT, с видимыми несовместимыми строками
indisready	bool		Если true, индекс готов к добавлению данных. Значение false означает, что индекс игнорируется операциями INSERT/UPDATE.
indislive	bool		Если false, индекс находится в процессе удаления и его следует игнорировать

Имя	Тип	Ссылки	Описание
			для любых целей (включая вопрос применимости HOT)
indisreplident	bool		Если true, этот индекс выбран в качестве «идентификатора реплики» командой ALTER TABLE ... REPLICA IDENTITY USING INDEX ...
indkey	int2vector	pg_attribute .attnum	Это массив из indnatts значений, указывающих, какие столбцы таблицы индексирует этот индекс. Например, значения 1 3 будут означать, что ключ индекса составляют первый и третий столбцы таблицы. Ноль в этом массиве означает, что соответствующий атрибут индекса определяется выражением со столбцами таблицы, а не просто ссылкой на столбец.
indcollation	oidvector	pg_collation .oid	Для каждого столбца в ключе индекса этот массив содержит OID правила сортировки для применения в этом индексе либо ноль, если тип данных этого столбца не сортируемый.
indclass	oidvector	pg_opclass .oid	Для каждого столбца в ключе индекса этот массив содержит OID применяемых классов операторов. Подробнее это рассматривается в описании pg_opclass .
indoption	int2vector		Это массив из indnatts значений, в которых хранятся битовые флаги для отдельных столбцов. Значение этих флагов определяется методом

Имя	Тип	Ссылки	Описание
			доступа конкретного индекса.
indexprs	pg_node_tree		Деревья выражений (в представлении nodeToString()) для атрибутов индекса, не являющихся простыми ссылками на столбцы. Этот список содержит один элемент для каждого нулевого значения в indkey. Значением может быть NULL, если все атрибуты индекса представляют собой простые ссылки.
indpred	pg_node_tree		Дерево выражения (в представлении nodeToString()) для предиката частичного индекса, либо NULL, если это не частичный индекс.

49.27. pg_inherits

В каталоге pg_inherits содержится информация об иерархиях наследования таблиц. Для каждой непосредственно дочерней таблицы в ней содержится одна запись. (Косвенное наследование можно определить, просмотрев цепочку записей.)

Таблица 49.27. Столбцы pg_inherits

Имя	Тип	Ссылки	Описание
inhrelid	oid	pg_class .oid	OID дочерней таблицы
inhparent	oid	pg_class .oid	OID родительской таблицы
inhseqno	int4		Если у дочерней таблицы есть несколько непосредственных родителей (множественное наследование), это число определяет порядок, в котором располагаются наследованные столбцы. Нумерация начинается с 1.

49.28. pg_init_privs

В каталоге pg_init_privs содержится информация об изначально назначаемых правах для объектов в системе. Для каждого объекта в базе данных, имеющего нестандартный (отличный от NULL) начальный набор прав, в ней содержится одна запись.

Начальные права доступа для объектов могут задаваться либо при инициализации базы данных (программой initdb), либо когда объект создаётся в процессе CREATE EXTENSION и скрипт расширения задаёт права, задействуя систему GRANT. Заметьте, что эта система автоматически записывает права, устанавливаемые скриптом расширения, так что авторам расширений достаточно использовать в своих скриптах только GRANT и REVOKE, чтобы права были сохранены. Столбец privtype показывает, были ли начальные права заданы программой initdb или в процессе выполнения команды CREATE EXTENSION.

Для объектов, которым начальные права были назначены программой initdb, записи в privtype помечаются буквой 'i', а для объектов, которым права назначались в процессе CREATE EXTENSION, — буквой 'e'.

Таблица 49.28. Столбцы pg_init_privs

Имя	Тип	Ссылки	Описание
objoid	oid	любой столбец OID	OID определённого объекта
classoid	oid	pg_class .oid	OID системного каталога, в котором находится объект
objsubid	int4		Для столбца таблицы это номер столбца (objoid и classoid указывают на саму таблицу). Для всех других типов объектов это поле содержит ноль.
privtype	char		Код, определяющий тип начального права для объекта; см. текст
initprivs	aclitem[]		Начальные права доступа; за подробностями обратитесь к описанию GRANT и REVOKE

49.29. pg_language

В каталоге pg_language регистрируются языки, на которых возможно писать функции или хранимые процедуры. За дополнительной информацией о языковых обработчиках обратитесь к описанию CREATE LANGUAGE и Главе 39.

Таблица 49.29. Столбцы pg_language

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
lanname	name		Имя языка
lanowner	oid	pg_authid .oid	Владелец языка
lanispl	bool		Для внутренних языков (например, SQL) содержит false, а для пользовательских

Имя	Тип	Ссылки	Описание
			языков — true. В настоящее время, pg_dump всё ещё пользуется этим признаком, чтобы определить, какие языки нужно выгружать, но в будущем ему на смену может прийти другой механизм.
lanpltrusted	bool		True, если это доверенный язык, что означает, что можно рассчитывать на то, что он не открывает доступ куда-либо за пределы обычной среды исполнения SQL. Создавать функции на недоверенных языках могут только суперпользователи.
lanplcallfoid	oid	pg_proc .oid	Для не внутренних языков это значение указывает на языковой обработчик, который представляет собой специальную функцию, отвечающую за выполнение всех процедур, написанных на этом языке
laninline	oid	pg_proc .oid	Это значение указывает на функцию, отвечающую за выполнение «внедрённых» анонимных блоков кода (блоков DO). Ноль, если внедрённые блоки не поддерживаются.
lanvalidator	oid	pg_proc .oid	Это значение указывает на функцию проверки языка, которая отвечает за проверку синтаксиса и правильности новых функций в момент их создания. Ноль, если функция проверки отсутствует.
lanacl	aclitem[]		Права доступа; за подробностями

Имя	Тип	Ссылки	Описание
			обратитесь к описанию GRANT и REVOKE

49.30. pg_largeobject

В каталоге pg_largeobject содержатся данные, образующие «большие объекты». Большой объект идентифицируется по OID, назначаемому при его создании. Каждый большой объект разделяется на сегменты или «страницы», достаточно небольшие для удобного размещения в строках таблицы pg_largeobject. Объём данных на странице определяется как LOBLKSIZE (что в настоящее время составляет BLCKSZ/4, то есть обычно 2 Кб).

До PostgreSQL 9.0 большие объекты не были связаны с механизмом разрешений. В результате таблица pg_largeobject была доступна на чтение для всех и через неё можно было получить OID (и содержимое) всех больших объектов в системе. Теперь это не так; для получения списка OID больших объектов нужно обратиться к pg_largeobject_metadata.

Таблица 49.30. Столбцы pg_largeobject

Имя	Тип	Ссылки	Описание
loid	oid	pg_largeobject_metadata .oid	Идентификатор большого объекта, включающего эту страницу
pageno	int4		Номер этой страницы в большом объекте (начиная с нуля)
data	bytea		Собственно данные, хранящиеся в большом объекте. Их объём не может превышать LOBLKSIZE, но может быть меньше.

В каждой строке pg_largeobject содержатся данные для одной строки большого объекта, начиная со смещения (pageno * LOBLKSIZE) внутри него (в байтах). Эта реализация допускает разреженное хранилище: страницы могут отсутствовать и могут быть короче LOBLKSIZE, даже если это не последние страницы объектов. Пропущенные области в большом объекте будут считываться как нулевые.

49.31. pg_largeobject_metadata

В каталоге pg_largeobject_metadata содержатся метаданные, связанные с большими объектами. Собственно данные больших объектов хранятся в pg_largeobject.

Таблица 49.31. Столбцы pg_largeobject_metadata

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
lomowner	oid	pg_authid .oid	Владелец большого объекта
lomacl	aclitem[]		Права доступа; за подробностями

Имя	Тип	Ссылки	Описание
			обратитесь к описанию GRANT и REVOKE

49.32. pg_namespace

В pg_namespace сохраняются пространства имён. Пространство имён представляет собой структуру, на которой основываются схемы SQL: в каждом пространстве имён без конфликтов может существовать отдельный набор отношений, типов и т. д.

Таблица 49.32. Столбцы pg_namespace

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
nspname	name		Имя пространства имён
nspowner	oid	pg_authid .oid	Владелец пространства имён
nspacl	aclitem[]		Права доступа; за подробностями обратитесь к описанию GRANT и REVOKE

49.33. pg_opclass

В каталоге pg_opclass определяются классы операторов для индексных методов доступа. Каждый класс операторов устанавливает конкретную операцию для индексируемых столбцов определённого типа данных и определённого метода доступа. Класс операторов по сути устанавливает, что некоторое семейство операторов применимо к определённому индексируемому типу столбца. Набор операторов из семейства, которые действительно можно использовать с индексируемым столбцом, образуют те, что принимают тип данных столбца в качестве левого операнда.

Классы операторов углублённо рассматриваются в Разделе 35.14.

Таблица 49.33. Столбцы pg_opclass

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
opcmethod	oid	pg_am .oid	Индексный метод доступа, для которого создан этот класс операторов
opcname	name		Имя этого класса операторов
opcnamespace	oid	pg_namespace .oid	Пространство имён этого класса операторов
opcowner	oid	pg_authid .oid	Владелец класса операторов

Имя	Тип	Ссылки	Описание
opcfamily	oid	pg_opfamily .oid	Семейство операторов, содержащее этот класс операторов
opcintype	oid	pg_type .oid	Тип данных, индексируемый данным классом операторов
opcdefault	bool		True, если этот класс операторов применяется по умолчанию для opcintype
opckeytype	oid	pg_type .oid	Тип данных, хранимых в индексе, или ноль, если он совпадает с opcintype

Значение opcmethod класса операторов должно совпадать с opfmethod для содержащего его семейства операторов. Кроме того, должно быть не больше одной строки в pg_opclass, в которой opcdefault равно true для любой данной комбинации opcmethod и opcintype.

49.34. pg_operator

В каталоге pg_operator хранится информация об операторах. За дополнительными сведениями обратитесь к описанию CREATE OPERATOR и Разделу 35.12.

Таблица 49.34. Столбцы pg_operator

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
oprname	name		Имя оператора
oprnamespace	oid	pg_namespace .oid	OID пространства имён, содержащего этот оператор
oprowner	oid	pg_authid .oid	Владелец оператора
oprkind	char		b = инфиксный («both»), l = префиксный («left»), r = постфиксный («right»)
oprcanmerge	bool		Этот оператор поддерживает соединение слиянием
oprcanhash	bool		Этот оператор поддерживает соединение по хешу
oprleft	oid	pg_type .oid	Тип левого операнда
oprright	oid	pg_type .oid	Тип правого операнда
oprresult	oid	pg_type .oid	Тип результата

Имя	Тип	Ссылки	Описание
oprcom	oid	pg_operator .oid	Коммутирующий для данного оператор, если есть
oprnegate	oid	pg_operator .oid	Обратный для данного оператор, если есть
oprcode	regproc	pg_proc .oid	Функция, реализующая этот оператор
oprrest	regproc	pg_proc .oid	Функция оценки избирательности ограничения для данного оператора
oprjoin	regproc	pg_proc .oid	Функция оценки избирательности соединения для данного оператора

Неиспользуемые поля содержат нули. Например, поле oprleft будет содержать ноль для префиксного оператора.

49.35. pg_opfamily

В каталоге pg_opfamily определяются семейства операторов. Каждое семейство операторов представляет собой набор операторов и связанных с ними опорных процедур, реализующих операции, требуемые для определённого индексного метода доступа. Более того, все операторы в семействе являются «совместимыми», в том смысле, который определяется методом доступа. Концепция семейства операторов позволяет применять в индексах операторы смешанных типов и рассматривать их, используя знание семантики метода доступа.

Семейства операторов углублённо рассматриваются в Разделе 35.14.

Таблица 49.35. Столбцы pg_opfamily

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
opfmethod	oid	pg_am .oid	Индексный метод доступа, для которого предназначено семейство операторов
opfname	name		Имя семейства операторов
opfnamespace	oid	pg_namespace .oid	Пространство имён семейства операторов
opfowner	oid	pg_authid .oid	Владелец семейства операторов

Основная часть информации, определяющей семейство операторов, находится не в строке pg_opfamily, а в связанных строках в pg_amop, pg_amproc и pg_opclass.

49.36. pg_partitioned_table

В каталоге pg_partitioned_table хранится информация о секционировании таблиц.

Таблица 49.36. Столбцы pg_partitioned_table

Имя	Тип	Ссылки	Описание
partrelid	oid	pg_class .oid	OID записи в pg_class для этой секционированной таблицы
partstrat	char		Стратегия секционирования; l = секционирование по списку (List), r = секционирование по диапазонам (Range)
partnatts	int2		Число столбцов в ключе разбиения
partattrs	int2vector	pg_attribute .attnum	Это массив из partnatts значений, указывающих, какие столбцы таблицы входят в ключ разбиения. Например, значения 1 3 будут означать, что ключ разбиения составляют первый и третий столбцы таблицы. Ноль в этом массиве означает, что соответствующей частью ключа разбиения является выражение, а не ссылка на отдельный столбец.
partclass	oidvector	pg_opclass .oid	Для каждого столбца в ключе разбиения этот массив содержит OID применяемых классов операторов. Подробнее это рассматривается в описании pg_opclass .
partcollation	oidvector	pg_opclass .oid	Для каждого столбца в ключе разбиения этот массив содержит OID правила сортировки для секционирования либо 0, если тип данных этого столбца не сортируемый.
partexprs	pg_node_tree		Деревья выражений (в представлении nodeToString()) для частей ключа разбиения, не являющихся простыми

Имя	Тип	Ссылки	Описание
			ссылками на столбцы. Этот список содержит один элемент для каждого нулевого значения в partattrs. Значением может быть NULL, если все части ключа разбиения являются простыми указаниями столбцов.

49.37. pg_pltemplate

В каталоге pg_pltemplate хранится информация о «шаблонах» для процедурных языков. Шаблон для языка позволяет создать язык в определённой базе данных простой командой CREATE LANGUAGE, без необходимости указывать подробности реализации.

В отличие от большинства системных каталогов, pg_pltemplate разделяется всеми базами данных в кластере: есть только один экземпляр pg_pltemplate в кластере, а не отдельные в базе данных. Благодаря этому, к данной информации при необходимости можно обращаться в любой базе данных.

Таблица 49.37. Столбцы pg_pltemplate

Имя	Тип	Описание
tmplname	name	Имя языка, для которого предназначен этот шаблон
tmpltrusted	boolean	True, если язык считается доверенным
tmpldbacreate	boolean	True, если язык может быть создан владельцем базы данных
tmplhandler	text	Имя функции-обработчика вызова
tmplinline	text	Имя функции-обработчика анонимного кода, либо NULL, если её нет
tmplvalidator	text	Имя функции проверки, либо NULL, если её нет
tmpllibrary	text	Путь к разделяемой библиотеке, реализующей этот язык
tmplacl	aclitem[]	Права доступа для шаблона (фактически не используются)

В настоящее время для управления шаблонами процедурных языков нет никаких команд; чтобы изменить встроенную информацию, суперпользователь должен модифицировать эту таблицу, выполняя обычные команды INSERT, DELETE или UPDATE.

> **Примечание**
>
> Скорее всего каталог pg_pltemplate будет ликвидирован в каком-нибудь будущем выпуске Postgres Pro, а эти знания о процедурных языках будут храниться в соответствующих скриптах установки расширений.

49.38. pg_policy

В каталоге pg_policy хранятся политики защиты на уровне строк для таблиц. Описание политики включает тип команды, к которой она применяется (это могут быть все команды), роли, к которым она применяется, выражение, добавляемое к условию барьера безопасности в запросы, обращающиеся к таблице, и выражение, добавляемое к условию WITH CHECK в запросы, которые пытаются добавить в таблицу новые записи.

Таблица 49.38. Столбцы pg_policy

Имя	Тип	Ссылки	Описание
polname	name		Имя политики.
polrelid	oid	pg_class .oid	Таблица, к которой применяется политика
polcmd	char		Тип команды, к которой применяется политика: r обозначает SELECT, a — INSERT, w — UPDATE, d — DELETE, а * — все команды
polpermissive	boolean		Является ли политика разрешительной или ограничительной?
polroles	oid[]	pg_authid .oid	Роли, к которым применяется политика.
polqual	pg_node_tree		Дерево выражения, добавляемое к условиям барьера безопасности в запросы, использующие таблицу
polwithcheck	pg_node_tree		Дерево выражения, добавляемое к условиям WITH CHECK в запросы, которые пытаются добавлять строки в таблицу

> **Примечание**
>
> Политики хранятся в pg_policy и применяются только когда для этой таблицы установлено свойство pg_class.relrowsecurity.

49.39. pg_proc

В каталоге pg_proc хранится информация о функциях (или процедурах). За дополнительными сведениями обратитесь к описанию CREATE FUNCTION и Разделу 35.3.

В этой таблице содержатся данные и агрегатных, и обычных функций. Если признак proisagg равен true, для этой функции должна быть ещё строка в pg_aggregate.

Таблица 49.39. Столбцы pg_proc

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
proname	name		Имя функции
pronamespace	oid	pg_namespace .oid	OID пространства имён, содержащего эту функцию
proowner	oid	pg_authid .oid	Владелец функции
prolang	oid	pg_language .oid	Язык реализации или интерфейс вызова для этой функции
procost	float4		Примерная стоимость выполнения (в единицах cpu_operator_cost); если установлен признак proretset, это стоимость выдачи одной строки
prorows	float4		Примерное число возвращаемых строк (ноль, если признак proretset не установлен)
provariadic	oid	pg_type .oid	Тип данных элементов переменного массива параметров, либо 0, если функция не принимает переменное число параметров
protransform	regproc	pg_proc .oid	Вызовы этой функции могут быть упрощены заданной функцией (см. Подраздел 35.9.10)
proisagg	bool		Функция является агрегатной
proiswindow	bool		Функция является оконной
prosecdef	bool		Функция определяет контекст безопасности (т. е. это функция «setuid»)
proleakproof	bool		Функция не имеет побочных эффектов. Никакая информация о её аргументах не выдаётся, кроме как через возвращаемое значение. Любая функция, которая

Имя	Тип	Ссылки	Описание
			может выдать ошибку, в зависимости от значений аргументов, не является герметичной.
proisstrict	bool		Функция возвращает NULL, если любой из аргументов при вызове NULL. В этом случае функция фактически не будет вызываться вовсе. Функции, не являющиеся «строгими», должны быть готовы принять значения NULL.
proretset	bool		Функция возвращает множество (т. е. множество значений указанного типа данных)
provolatile	char		Свойство provolatile говорит, зависит ли результат функции только от её входных аргументов, либо на него влияют внешние факторы. Буквой i обозначаются постоянные функции («immutable»), которые всегда возвращают один результат для одних и тех же аргументов. Буквой s обозначаются стабильные функции («stable»), результаты которых (для одних и тех же аргументов) не меняются в ходе одного сканирования. Буквой v обозначаются изменчивые функции («volatile»), результаты которых могут меняться в любое время. (Так же v следует выбирать для функций с побочными эффектами, чтобы система не оптимизировала их вызовы.)

Имя	Тип	Ссылки	Описание
proparallel	char		Свойство proparallel говорит, может ли эта функция безопасно выполняться в параллельном режиме. Символом s в нём обозначаются функции, которые могут выполняться в параллельном режиме без ограничений. Символом r обозначаются функции, которые могут выполняться в параллельном режиме, но только в ведущем процессе группы; в параллельных рабочих процессах вызывать их нельзя. Символом u отмечаются функции небезопасные в параллельном режиме; присутствие такой функции влечёт выбор последовательного плана выполнения запроса.
pronargs	int2		Число входных аргументов
pronargdefaults	int2		Число аргументов, для которых определены значения по умолчанию
prorettype	oid	pg_type .oid	Тип данных возвращаемого значения
proargtypes	oidvector	pg_type .oid	Массив с типами данных аргументов функции. В нём учитываются только входные аргументы функции (включая аргументы INOUT и VARIADIC), так что он представляет сигнатуру вызова функции.
proallargtypes	oid[]	pg_type .oid	Массив с типами данных аргументов функции. В нём учитываются все аргументы (включая

Имя	Тип	Ссылки	Описание
			аргументы OUT и INOUT); однако если все аргументы только входные (IN), это поле будет содержать NULL. Заметьте, что индексы в нём начинаются с 1, тогда как в proargtypes по историческим причинам они начинаются с 0.
proargmodes	char[]		Массив с режимами аргументов функций, закодированными как i для входных аргументов (IN), o для выходных аргументов (OUT), b для аргументов входных и выходных одновременно (INOUT), v для переменных аргументов (VARIADIC) и t для табличных аргументов (TABLE). Если все аргументы являются аргументами IN, это поле может содержать NULL. Заметьте, что индексы в этом массиве соответствуют позициям в proallargtypes, а не в proargtypes.
proargnames	text[]		Массив с именами аргументов функции. Для аргументов без имени в этом массиве задаются пустые строки. Если все аргументы функции безымянные, это поле может содержать NULL. Заметьте, что индексы в этом массиве соответствуют позициям в proallargtypes, а не в proargtypes.
proargdefaults	pg_node_tree		Деревья выражений (в представлении nodeToString()) для значений аргументов по умолчанию. Это

Имя	Тип	Ссылки	Описание
			список, содержащий pronargdefaults элементов, соответствующих последним *N* *входным* аргументам (т. е., последним *N* позициям в proargtypes). Если значение по умолчанию не имеет никакой аргумент, это поле может содержать NULL.
protrftypes	oid[]		OID типов данных, к которым будут применяться трансформации.
prosrc	text		Это значение говорит обработчику функции, как вызывать данную функцию. Это может быть собственно исходный код функции для интерпретируемых языков, объектный символ, имя файла или что-то другое, в зависимости от языка реализации/ соглашения о вызовах.
probin	text		Дополнительная информация о том, как вызывать функцию. Интерпретация этого значения так же зависит от языка.
proconfig	text[]		Локальные присваивания конфигурационных переменных времени выполнения, действующие в рамках функции
proacl	aclitem[]		Права доступа; за подробностями обратитесь к описанию GRANT и REVOKE

Для скомпилированных функций, как встроенных, так и динамически загружаемых, поле prosrc содержит имя функции на языке C (объектный символ). Для всех других известных сегодня языков поле prosrc содержит исходный код функции. Поле probin используется только для динамически загружаемых функций на C, для которых оно задаёт имя разделяемой библиотеки, содержащей эти функции.

49.40. pg_publication

Каталог pg_publication содержит все публикации, созданные в базе данных. Подробнее о публикациях можно узнать в Разделе 30.1.

Таблица 49.40. Столбцы pg_publication

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
pubname	name		Имя публикации
pubowner	oid	pg_authid .oid	Владелец публикации
puballtables	bool		Если true, эта публикация автоматически включает все таблицы в базе данных, в том числе и те, что будут созданы в будущем.
pubinsert	bool		Если true, операции INSERT реплицируются для таблиц в репликации.
pubupdate	bool		Если true, операции UPDATE реплицируются для таблиц в публикации.
pubdelete	bool		Если true, операции DELETE реплицируются для таблиц в публикации.

49.41. pg_publication_rel

Каталог pg_publication_rel содержит сопоставления отношений и публикаций в базе данных (это сопоставления вида многие-ко-многим). Более понятное пользователю представление этой информации можно также получить в Разделе 49.78.

Таблица 49.41. Столбцы pg_publication_rel

Имя	Тип	Ссылки	Описание
prpubid	oid	pg_publication .oid	Ссылка на публикацию
prrelid	oid	pg_class .oid	Ссылка на отношение

49.42. pg_range

В каталоге pg_range хранится информация о типах диапазонов. Эта информация дополняет записи типов в pg_type.

Таблица 49.42. Столбцы pg_range

Имя	Тип	Ссылки	Описание
rngtypid	oid	pg_type .oid	OID типа диапазона

Имя	Тип	Ссылки	Описание
rngsubtype	oid	pg_type .oid	OID типа элемента (подтипа) данного типа диапазона
rngcollation	oid	pg_collation .oid	OID правила сортировки, применяемого для сравнения диапазонов, либо 0 в случае его отсутствия
rngsubopc	oid	pg_opclass .oid	OID класса операторов подтипа, применяемого для сравнения диапазонов
rngcanonical	regproc	pg_proc .oid	OID функции, преобразующей значение диапазона в каноническую форму, либо 0 в случае её отсутствия
rngsubdiff	regproc	pg_proc .oid	OID функции, возвращающей разницу между значениями двух элементов в значении double precision, либо 0 в случае её отсутствия

Значение rngsubopc (в сочетании с rngcollation, если тип элемента сортируемый) определяет порядок сортировки для типа диапазона. Значение rngcanonical используется, когда тип элемента дискретный. Значение rngsubdiff может отсутствовать, но его рекомендуется задавать для увеличения производительности индексов GiST с диапазонным типом.

49.43. pg_replication_origin

В каталоге pg_replication_origin содержатся все созданные источники репликации. Подробно источники репликации описаны в Главе 47.

Таблица 49.43. Столбцы pg_replication_origin

Имя	Тип	Ссылки	Описание
roident	Oid		Уникальный в рамках кластера идентификатор для источника репликации. Не должен покидать пределы системы.
roname	text		Определяемое пользователем внешнее имя источника репликации.

49.44. pg_rewrite

В каталоге pg_rewrite хранятся правила перезаписи для таблиц и представлений.

Таблица 49.44. Столбцы pg_rewrite

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
rulename	name		Имя правила
ev_class	oid	pg_class .oid	Таблица, к которой относится это правило
ev_type	char		Тип события, которое обрабатывается этим правилом: 1 = SELECT, 2 = UPDATE, 3 = INSERT, 4 = DELETE
ev_enabled	char		Устанавливает, в каких режимах session_replication_role срабатывает правило: O = правило срабатывает в режимах «origin» (источник) и «local» (локально), D = правило отключено, R = правило срабатывает в режиме «replica» (реплика), A = правило срабатывает всегда.
is_instead	bool		True, если это правило INSTEAD
ev_qual	pg_node_tree		Дерево выражения (в форме представления nodeToString()) для условия применения правила
ev_action	pg_node_tree		Дерево запроса (в форме представления nodeToString()) для действия правила

> **Примечание**
>
> Если для таблицы есть какие-либо правила в этом каталоге, значением pg_class.relhasrules для неё должно быть true.

49.45. pg_seclabel

В каталоге pg_seclabel хранятся метки безопасности для объектов баз данных. Управлять метками безопасности можно с помощью команды SECURITY LABEL. Более простой способ просмотра меток безопасности описан в Разделе 49.83.

Также обратите внимание на каталог pg_shseclabel, который выполняет ту же функцию для меток безопасности глобальных объектов в кластере баз данных.

Таблица 49.45. Столбцы pg_seclabel

Имя	Тип	Ссылки	Описание
objoid	oid	любой столбец OID	OID объекта, к которому относится эта метка безопасности
classoid	oid	pg_class .oid	OID системного каталога, к которому относится этот объект
objsubid	int4		Для метки безопасности, связанной со столбцом таблицы, это номер столбца (objoid и classoid указывают на саму таблицу). Для всех других типов объектов это поле содержит ноль.
provider	text		Поставщик меток безопасности, связанный с этой меткой.
label	text		Метка безопасности, применённая к этому объекту.

49.46. pg_sequence

В каталоге pg_sequence содержится информация о последовательностях. Некоторая информация о последовательностях, в частности имя и схема, хранится в pg_class.

Таблица 49.46. Столбцы pg_sequence

Имя	Тип	Ссылки	Описание
seqrelid	oid	pg_class .oid	OID записи в pg_class для этой последовательности
seqtypid	oid	pg_type .oid	Тип данных последовательности
seqstart	int8		Начальное значение последовательности
seqincrement	int8		Шаг увеличения последовательности
seqmax	int8		Максимальное значение последовательности
seqmin	int8		Минимальное значение последовательности
seqcache	int8		Размер кеша последовательности
seqcycle	bool		Зацикливается ли последовательность

49.47. pg_shdepend

В каталоге pg_shdepend записываются отношения зависимости между объектами баз данных и разделяемыми объектами, такими как роли. Эта информация позволяет Postgres Pro удостовериться, что эти объекты не используется, прежде чем удалять их.

Также смотрите каталог pg_depend, который играет подобную роль в отношении зависимостей объектов в одной базе данных.

В отличие от большинства системных каталогов, pg_shdepend разделяется всеми базами данных кластера: есть только один экземпляр pg_shdepend в кластере, а не отдельные в каждой базе данных.

Таблица 49.47. Столбцы pg_shdepend

Имя	Тип	Ссылки	Описание
dbid	oid	pg_database .oid	OID базы данных, в которой находится зависимый объект, или ноль, если это глобальный объект
classid	oid	pg_class .oid	OID системного каталога, в котором находится зависимый объект
objid	oid	любой столбец OID	OID определённого зависимого объекта
objsubid	int4		Для столбца таблицы это номер столбца (objid и classid указывают на саму таблицу). Для всех других типов объектов это поле содержит ноль.
refclassid	oid	pg_class .oid	OID системного каталога, к которому относится вышестоящий объект (это должен быть разделяемый каталог)
refobjid	oid	любой столбец OID	OID определённого вышестоящего объекта
deptype	char		Код, определяющий конкретную семантику данного отношения зависимости; см. текст

Во всех случаях, запись в pg_shdepend показывает, что вышестоящий объект нельзя удалить, не удаляя подчинённый объект. Однако есть несколько подвидов зависимости, задаваемых в поле deptype:

SHARED_DEPENDENCY_OWNER (o)

Вышестоящий объект (это должна быть роль) является владельцем зависимого объекта.

SHARED_DEPENDENCY_ACL (a)

Вышестоящий объект (это должна быть роль) упоминается в ACL (списке управления доступом, то есть списке прав) подчинённого объекта. (Запись SHARED_DEPENDENCY_ACL не создаётся для владельца объекта, так как для владельца всё равно имеется запись SHARED_DEPENDENCY_OWNER.)

SHARED_DEPENDENCY_POLICY (r)

Вышестоящий объект (это должна быть роль) упомянут в качестве целевого в объекте зависимой политики.

SHARED_DEPENDENCY_PIN (p)

Зависимый объект отсутствует; этот тип записи показывает, что система сама зависит от вышестоящего объекта, так что этот объект нельзя удалять ни при каких условиях. Записи этого типа создаются только командой initdb. Поля зависимого объекта в такой записи содержат нули.

В будущем могут появиться и другие подвиды зависимости. Заметьте в частности, что с текущим определением вышестоящими объектами могут быть только роли.

49.48. pg_shdescription

В каталоге pg_shdescription хранятся дополнительные описания (комментарии) для глобальных объектов баз данных. Описания можно задавать с помощью команды COMMENT и просматривать в psql, используя команды \d.

Также смотрите каталог pg_description, который играет подобную роль в отношении описаний объектов в одной базе данных.

В отличие от большинства системных каталогов, pg_shdescription разделяется всеми базами данных кластера: есть только один экземпляр pg_shdescription в кластере, а не отдельные в каждой базе данных.

Таблица 49.48. Столбцы pg_shdescription

Имя	Тип	Ссылки	Описание
objoid	oid	любой столбец OID	OID объекта, к которому относится это описание
classoid	oid	pg_class .oid	OID системного каталога, к которому относится этот объект
description	text		Произвольный текст, служащий описанием данного объекта

49.49. pg_shseclabel

В каталоге pg_shseclabel хранятся метки безопасности для глобальных объектов баз данных. Управлять метками безопасности можно с помощью команды SECURITY LABEL. Более простой способ просмотра меток безопасности описан в Разделе 49.83.

Также смотрите каталог pg_seclabel, который играет подобную роль в отношении меток безопасности объектов в одной базе данных.

В отличие от большинства системных каталогов, pg_shseclabel разделяется всеми базами данных кластера: есть только один экземпляр pg_shseclabel в кластере, а не отдельные в каждой базе данных.

Таблица 49.49. Столбцы pg_shseclabel

Имя	Тип	Ссылки	Описание
objoid	oid	любой столбец OID	OID объекта, к которому относится эта метка безопасности
classoid	oid	pg_class .oid	OID системного каталога, к которому относится этот объект
provider	text		Поставщик меток безопасности, связанный с этой меткой.
label	text		Метка безопасности, применённая к этому объекту.

49.50. pg_statistic

В каталоге pg_statistic хранится статистическая информация о содержимом базы данных. Записи в нём создаются командой ANALYZE, а затем используются планировщиком запросов. Заметьте, что все эти данные по природе своей неточные, даже если предполагается, что они актуальны.

Обычно для каждого столбца, подлежащего анализу, в этом каталоге есть одна запись со значением stainherit = false. Если у таблицы имеются потомки в иерархии наследования, также создаётся вторая запись с stainherit = true. Эта строка представляет статистику по столбцу в дереве наследования, то есть статистику по данным, которые возвратит запрос SELECT *столбец* FROM *таблица**, тогда как строка с stainherit = false представляет результаты запроса SELECT *столбец* FROM ONLY *таблица*.

В pg_statistic также хранится статистическая информация о значениях выражений индексов. Она описывается так же, как если бы это были столбцы данных; в частности, starelid ссылается на индекс. Однако для столбцов, задействуемых в индексе без выражений, дополнительная запись не добавляется, так как она повторяла бы запись для нижележащего столбца таблицы. В настоящее время во всех записях для выражений индексов stainherit = false.

Так как для различных типов данных могут быть уместны различные типы статистики, в каталоге pg_statistic не делается конкретных предположений о том, какая статистика в нём хранится. Отдельные столбцы в pg_statistic выделены только для самых общих свойств (например, доля NULL). Всё остальное хранится в «слотах», представляющих собой группы связанных столбцов, содержимое которых определяется кодовым числом в одном из столбцов слотов. За подробностями обратитесь к src/include/catalog/pg_statistic.h.

Каталог pg_statistic не должен быть доступен на чтение всем, так как даже статистическая информация о содержимом таблицы может считаться конфиденциальной. (Например, довольно интересны могут быть минимальные и максимальные значения в столбце зарплаты.) Поэтому существует pg_stats — доступное всем для чтения представление на базе pg_statistic, в котором выдаётся информация только по тем таблицам, которые может читать текущий пользователь.

Таблица 49.50. Столбцы pg_statistic

Имя	Тип	Ссылки	Описание
starelid	oid	pg_class .oid	Таблица (или индекс), к которой принадлежит описываемый столбец
staattnum	int2	pg_attribute .attnum	Номер описываемого столбца

Имя	Тип	Ссылки	Описание
stainherit	bool		Если true, в статистике учитываются значения в дочерних столбцах, а не только в указанном отношении
stanullfrac	float4		Доля записей, в которых этот столбец содержит NULL
stawidth	int4		Средний размер хранения не NULL-элементов, в байтах
stadistinct	float4		Число различных и отличных от NULL значений в столбце. Число больше нуля представляет фактическое количество различных значений. Если это число меньше нуля, его модуль представляет множитель для общего количества строк в таблице; например, для столбца, в котором примерно 80% значений не NULL, и каждое отличное от NULL значение в среднем повторяется дважды, может быть представлено значение stadistinct = -0.4. Ноль означает, что количество различных значений неизвестно.
stakind*N*	int2		Кодовое число, определяющее род статистики, хранящейся в *N*-ом «слоте» строки pg_statistic row.
staop*N*	oid	pg_operator .oid	Оператор, с которым была получена статистика, хранящаяся в *N*-ом «слоте». Например, для слота гистограммы это будет оператор <, определяющий порядок сортировки данных.
stanumbers*N*	float4[]		Численная статистика соответствующего рода

Имя	Тип	Ссылки	Описание
			для *N*-ного «слота», либо NULL, если с этим родом слота не связаны числовые значения
stavalues*N*	anyarray		Значения столбцов соответствующего рода для *N*-го «слота», либо NULL, если для этого рода слота не хранятся никакие значения. Все значения элементов массива фактически имеют тип данных столбца или связанный тип, например, тип элемента массива, так что определить типы эти столбцов более конкретно, чем anyarray, нельзя.

49.51. pg_statistic_ext

Каталог pg_statistic_ext содержит расширенную статистику планировщика. Каждая строка в этом каталоге соответствует *объекту статистики*, созданному командой CREATE STATISTICS.

Таблица 49.51. Столбцы pg_statistic_ext

Имя	Тип	Ссылки	Описание
stxrelid	oid	pg_class .oid	Таблица, содержащая столбцы, описываемые этим объектом
stxname	name		Имя объекта статистики
stxnamespace	oid	pg_namespace .oid	OID пространства имён, содержащего этот объект статистики
stxowner	oid	pg_authid .oid	Владелец объекта статистики
stxkeys	int2vector	pg_attribute .attnum	Массив номеров атрибутов, показывающий, какие столбцы таблицы покрываются данным объектом статистики; например, значение 1 3 показывает, что статистика покрывает первый и третий столбцы таблицы
stxkind	char[]		Массив, содержащий коды для включённых видов статистики; допустимые значения:

Имя	Тип	Ссылки	Описание
			d для статистики по количеству различных значений, f для статистики по функциональным зависимостям
stxndistinct	pg_ndistinct		Количество различных значений, сериализованное в типе pg_ndistinct
stxdependencies	pg_dependencies		Статистика по функциональным зависимостям, сериализованная в типе pg_dependencies

Поле stxkind заполняется при создании объекта статистики и показывает, для какого типа(ов) статистики он создан. Поля после него изначально содержат NULL и заполняются только когда соответствующая статистика вычисляется командой ANALYZE.

49.52. pg_subscription

В каталоге pg_subscription содержатся все существующие подписки логической репликации. Подробнее логическая репликация описана в Главе 30.

В отличие от большинства системных каталогов, pg_subscription разделяется всеми базами данных кластера: есть только один экземпляр pg_subscription в кластере, а не отдельные в каждой базе данных.

Обычные пользователи не имеют доступа к столбцу subconninfo, так как он может содержать пароль в открытом виде.

Таблица 49.52. Столбцы pg_subscription

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
subdbid	oid	pg_database .oid	OID базы данных, в которой располагается эта подписка
subname	name		Имя подписки
subowner	oid	pg_authid .oid	Владелец подписки
subenabled	bool		Если true, подписка включена и должна реплицироваться.
subsynccommit	text		Содержит значение параметра synchronous_commit для рабочих процессов подписки.

Имя	Тип	Ссылки	Описание
subconninfo	text		Строка подключения к вышестоящей базе данных
subslotname	name		Имя слота репликации в вышестоящей базе данных (также применяется в качестве локального имени источника репликации); значение null соответствует имени NONE
subpublications	text[]		Массив имён публикаций, на которые оформлена подписка. Подписки с этими именами должны быть опубликованы на сервере. Подробнее публикации описаны в Разделе 30.1.

49.53. pg_subscription_rel

Каталог pg_subscription_rel содержит состояние каждого реплицируемого отношения в каждой подписке (это связь вида многие-ко-многим).

Этот каталог содержит только таблицы, известные в подписке после выполнения CREATE SUBSCRIPTION или ALTER SUBSCRIPTION ... REFRESH PUBLICATION.

Таблица 49.53. Столбцы pg_subscription_rel

Имя	Тип	Ссылки	Описание
srsubid	oid	pg_subscription .oid	Ссылка на подписку
srrelid	oid	pg_class .oid	Ссылка на отношение
srsubstate	char		Код состояния: i = инициализация, d = копирование данных, s = синхронизация выполнена, r = готовность (обычная репликация)
srsublsn	pg_lsn		LSN изменения состояния на удалённой стороне, который используются для координации синхронизации в состояниях s или r; в других случаях — null

49.54. pg_tablespace

В каталоге pg_tablespace хранится информация о доступных табличных пространствах. Таблицы могут быть распределены по разным табличным пространствам для эффективного использования дискового хранилища.

В отличие от большинства системных каталогов, pg_tablespace разделяется всеми базами данных кластера: есть только один экземпляр pg_tablespace в кластере, а не отдельные для каждой базы данных.

Таблица 49.54. Столбцы pg_tablespace

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
spcname	name		Имя табличного пространства
spcowner	oid	pg_authid .oid	Владелец табличного пространства, обычно пользователь, создавший его
spcacl	aclitem[]		Права доступа; за подробностями обратитесь к описанию GRANT и REVOKE
spcoptions	text[]		Параметры уровня табличного пространства, в виде строк «ключ=значение»

49.55. pg_transform

В каталоге pg_transform хранятся сведения о трансформациях, которые представляют собой механизм адаптирования типов данных для процедурных языков. За дополнительной информацией обратитесь к CREATE TRANSFORM.

Таблица 49.55. Столбцы pg_transform

Имя	Тип	Ссылки	Описание
trftype	oid	pg_type .oid	OID типа данных, для которого предназначена трансформация
trflang	oid	pg_language .oid	OID языка, для которого предназначена трансформация
trffromsql	regproc	pg_proc .oid	OID функции, вызываемой для преобразования типа данных, подаваемого на вход процедурному языку (например, в параметрах функции). Ноль, если эта

Имя	Тип	Ссылки	Описание
			операция не поддерживается.
trftosql	regproc	pg_proc .oid	OID функции, вызываемой для преобразования значения, выдаваемого процедурным языком, (например, возвращаемых значений) к типу данных. Ноль, если эта операция не поддерживается.

49.56. pg_trigger

В каталоге pg_trigger хранятся триггеры для таблиц и представлений. За дополнительными сведениями обратитесь к описанию CREATE TRIGGER.

Таблица 49.56. Столбцы pg_trigger

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
tgrelid	oid	pg_class .oid	Таблица, к которой относится этот триггер
tgname	name		Имя триггера (должно быть уникальным среди триггеров одной таблицы)
tgfoid	oid	pg_proc .oid	Вызываемая функция
tgtype	int2		Битовая маска, задающая условия срабатывания триггера
tgenabled	char		Устанавливает, в каких режимах session_replication_role срабатывает триггер: O = триггер срабатывает в режимах «origin» (источник) и «local» (локально), D = триггер отключён, R = триггер срабатывает в режиме «replica» (реплика), A = триггер срабатывает всегда.
tgisinternal	bool		True, если триггер создан внутри системы (обычно, для реализации ограничения,

Имя	Тип	Ссылки	Описание
			заданного в tgconstraint)
tgconstrrelid	oid	pg_class .oid	Таблица, задействованная в ограничении ссылочной целостности
tgconstrindid	oid	pg_class .oid	Индекс, поддерживающий ограничение уникальности, первичного ключа или ссылочной целостности, либо ограничение-исключение
tgconstraint	oid	pg_constraint .oid	Запись в pg_constraint , связанная этим триггером, если такая имеется
tgdeferrable	bool		True, если триггер ограничения является откладываемым
tginitdeferred	bool		True, если триггер ограничения изначально отложенный
tgnargs	int2		Число аргументов, передаваемых функции триггера
tgattr	int2vector	pg_attribute .attnum	Номера столбцов, если триггер привязан к столбцам; в противном случае пустой массив
tgargs	bytea		Аргументы строкового типа, передаваемые триггеру, с NULL в конце каждого
tgqual	pg_node_tree		Дерево выражения (в представлении nodeToString()) для условия триггера WHEN, либо NULL, если оно отсутствует
tgoldtable	name		Предложение REFERENCING для OLD TABLE или NULL в случае его отсутствия
tgnewtable	name		Предложение REFERENCING для NEW TABLE или NULL в случае его отсутствия

В настоящее время триггеры, привязанные к столбцам, поддерживаются только для событий UPDATE, так что tgattr применимо только к событиям такого типа. Поле tgtype может содержать биты и для других типов событий, но они распространяются только на таблицы, вне зависимости от значения tgattr.

> **Примечание**
>
> Когда tgconstraint содержит не ноль, то есть ссылается на запись в pg_constraint, поля tgconstrrelid, tgconstrindid, tgdeferrable и tginitdeferred по большому счёту избыточны, они повторяют значения в этой записи. Однако возможно связать неоткладываемый триггер с откладываемым ограничением: с ограничениями внешнего ключа могут быть связаны и откладываемые, и неоткладываемые триггеры.

> **Примечание**
>
> Если для отношения есть какие-либо триггеры в этом каталоге, значением pg_class.relhastriggers для неё должно быть true.

49.57. pg_ts_config

Каталог pg_ts_config содержит записи, представляющие конфигурации текстового поиска. Конфигурация задаёт определённый анализатор текстового поиска и список словарей, которые будут использоваться для каждого из фрагментов, выдаваемых анализатором. Анализатор задаётся в записи в pg_ts_config, а сопоставления фрагментов со словарями определяются в подчинённых записях в pg_ts_config_map.

Возможности текстового поиска Postgres Pro углублённо рассматриваются в Главе 12.

Таблица 49.57. Столбцы pg_ts_config

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
cfgname	name		Имя конфигурации текстового поиска
cfgnamespace	oid	pg_namespace .oid	OID пространства имён, содержащего эту конфигурацию
cfgowner	oid	pg_authid .oid	Владелец конфигурации
cfgparser	oid	pg_ts_parser .oid	OID анализатора текстового поиска для этой конфигурации

49.58. pg_ts_config_map

В каталоге pg_ts_config_map содержатся записи, показывающие, к каким словарям текстового поиска и в каком порядке следует обращаться для каждого типа фрагмента, выдаваемого каждым анализатором текстового поиска.

Возможности текстового поиска Postgres Pro углублённо рассматриваются в Главе 12.

Таблица 49.58. Столбцы pg_ts_config_map

Имя	Тип	Ссылки	Описание
mapcfg	oid	pg_ts_config .oid	OID записи в pg_ts_config, к которой относится эта запись сопоставления
maptokentype	integer		Тип фрагмента, выдаваемый анализатором текстового поиска
mapseqno	integer		Порядок, в котором нужно просматривать эту запись (меньшие mapseqno просматриваются сначала)
mapdict	oid	pg_ts_dict .oid	OID словаря текстового поиска, к которому нужно обратиться

49.59. pg_ts_dict

В каталоге pg_ts_dict содержатся записи, определяющие словари текстового поиска. Словарь зависит от шаблона текстового поиска, в котором задаются все требуемые функции реализации; сам словарь предоставляет значения для настраиваемых параметров, поддерживаемых шаблонов. Такое разделение позволяет создавать словари непривилегированным пользователям. Параметры задаются текстовой строкой dictinitoption, их формат и значение зависят от шаблона.

Возможности текстового поиска Postgres Pro углублённо рассматриваются в Главе 12.

Таблица 49.59. Столбцы pg_ts_dict

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
dictname	name		Имя словаря текстового поиска
dictnamespace	oid	pg_namespace .oid	OID пространства имён, содержащего этот словарь
dictowner	oid	pg_authid .oid	Владелец словаря
dicttemplate	oid	pg_ts_template .oid	OID шаблона текстового поиска для этого словаря
dictinitoption	text		Строка с параметрами инициализации для шаблона

49.60. pg_ts_parser

В каталоге pg_ts_parser содержатся записи, определяющие анализаторы текстового поиска. Анализатор отвечает за разделение входного текста на лексемы и назначение типа фрагмента

каждой лексеме. Так как анализатор должен быть реализован в функции на языке уровня C, создавать новые анализаторы разрешено только суперпользователям баз данных.

Возможности текстового поиска Postgres Pro углублённо рассматриваются в Главе 12.

Таблица 49.60. Столбцы `pg_ts_parser`

Имя	Тип	Ссылки	Описание
`oid`	`oid`		Идентификатор строки (скрытый атрибут; должен выбираться явно)
`prsname`	`name`		Имя анализатора текстового поиска
`prsnamespace`	`oid`	`pg_namespace .oid`	OID пространства имён, содержащего этот анализатор
`prsstart`	`regproc`	`pg_proc .oid`	OID функции запуска анализатора
`prstoken`	`regproc`	`pg_proc .oid`	OID функции анализатора, выдающей следующий фрагмент
`prsend`	`regproc`	`pg_proc .oid`	OID функции анализатора, оканчивающей разбор
`prsheadline`	`regproc`	`pg_proc .oid`	OID функции анализатора, выдающей выдержки
`prslextype`	`regproc`	`pg_proc .oid`	OID функции анализатора лексических типов

49.61. `pg_ts_template`

В каталоге `pg_ts_template` содержатся записи, определяющие шаблоны текстового поиска. Шаблон представляет собой заготовку для класса словарей текстового поиска. Так как шаблон должен быть реализован в функциях на уровне языка C, создавать новые шаблоны разрешено только суперпользователям базы.

Возможности текстового поиска Postgres Pro углублённо рассматриваются в Главе 12.

Таблица 49.61. Столбцы `pg_ts_template`

Имя	Тип	Ссылки	Описание
`oid`	`oid`		Идентификатор строки (скрытый атрибут; должен выбираться явно)
`tmplname`	`name`		Имя шаблона текстового поиска
`tmplnamespace`	`oid`	`pg_namespace .oid`	OID пространства имён, содержащего этот шаблон

Имя	Тип	Ссылки	Описание
tmplinit	regproc	pg_proc .oid	OID функции инициализации шаблона
tmpllexize	regproc	pg_proc .oid	OID функции выделения лексем

49.62. pg_type

В каталоге pg_type хранится информация о типах данных. Базовые типы и типы-перечисления (скалярные типы) создаются командой CREATE TYPE, а домены — командой CREATE DOMAIN. При добавлении любой таблицы в базу данных автоматически создаётся составной тип, представляющий структуру строки таблицы. Также возможно создавать составные типы с помощью команды CREATE TYPE AS.

Таблица 49.62. Столбцы pg_type

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
typname	name		Имя типа данных
typnamespace	oid	pg_namespace .oid	OID пространства имён, содержащего этот тип
typowner	oid	pg_authid .oid	Владелец типа
typlen	int2		Для типа фиксированного размера в typlen задаётся число байт во внутреннем представлении типа. Но для типов переменной длины, typlen будет отрицательным. Значение -1 обозначает тип «varlena» (он содержит машинное слово, определяющее длину), а -2 обозначает строку в стиле C, оканчивающуюся нулём.
typbyval	bool		Поле typbyval определяет, будут ли внутренние процедуры передавать переменные этого типа по значению или по ссылке. Полю typbyval лучше присвоить false, если длина typlen не равна 1, 2 или

Имя	Тип	Ссылки	Описание
			4 (либо 8, на 64-битных машинах). Типы переменной длины всегда передаются по ссылке. Заметьте, что typbyval может быть false, даже если размер типа позволяет передачу по значению.
typtype	char		Поле typtype принимает значение b для базового типа (base), c для составного (composite), то есть типа строки таблицы, d для домена (domain), e для перечисления (enum), p для псевдотипа (pseudo-type) или r для диапазона (range). См. также typrelid и typbasetype.
typcategory	char		В поле typcategory задаётся произвольная классификация типов данных, на основе которой анализатор запросов может определить, какие неявные приведения будут «предпочитаемыми». См. Таблицу 49.63.
typispreferred	bool		True, если этот тип является предпочитаемым целевым типом в своей категории (typcategory)
typisdefined	bool		True, если тип определён, и false, если это тип-заготовка для ещё не определённого типа. Когда значение typisdefined — false, можно полагаться только на заданное имя, пространство имён и OID типа.
typdelim	char		Символ, разделяющий два значения этого типа при разборе вводимого массива. Заметьте, что этот

Имя	Тип	Ссылки	Описание
			разделитель связывается с типом данных элемента массива, а не с типом самого массива.
typrelid	oid	pg_class .oid	Если это составной тип (см. typtype), этот столбец указывает на запись pg_class , определяющую соответствующую таблицу. (Для независимого составного типа запись в pg_class на самом деле не представляет таблицу, но она всё равно нужна для связывания с записями pg_attribute этого типа.) Для не составных типов содержит ноль.
typelem	oid	pg_type .oid	Если значение typelem не 0, оно указывает на другую строку в pg_type . В этом случае к текущему типу можно обращаться по индексу, как к массиву, и получать значения типа typelem. «Настоящий» тип массива имеет переменную длину (typlen = -1), но для некоторых типов фиксированной длины (typlen > 0) также определяется typelem, например, для name и point. Если для типа фиксированной длины определён typelem, его внутренним представлением будет некоторое количество значений типа typelem, без других данных. Типы массивов переменной длины также содержат заголовок, определяемый подпрограммами массива.

Имя	Тип	Ссылки	Описание
typarray	oid	pg_type .oid	Если поле typarray не равно 0, оно указывает на другую запись в pg_type , описывающую «настоящий» тип массива, в которой этот тип будет элементом
typinput	regproc	pg_proc .oid	Функция преобразования ввода (из текстового формата)
typoutput	regproc	pg_proc .oid	Функция преобразования вывода (в текстовый формат)
typreceive	regproc	pg_proc .oid	Функция преобразования ввода (из двоичного формата), либо 0, если её нет
typsend	regproc	pg_proc .oid	Функция преобразования вывода (в двоичный формат), либо 0, если её нет
typmodin	regproc	pg_proc .oid	Функция ввода модификатора типа, либо 0, если тип не поддерживает модификаторы
typmodout	regproc	pg_proc .oid	Функция вывода модификатора типа, либо 0 для использования стандартного формата
typanalyze	regproc	pg_proc .oid	Нестандартная функция ANALYZE, либо 0 для использования стандартной функции
typalign	char		Переменная typalign определяет выравнивание, требуемое при хранении значения этого типа. Эта величина применяется при хранении на диске, а также для большинства представлений значений внутри Postgres Pro. Когда последовательно хранятся несколько значений, как например в представлении полной

Имя	Тип	Ссылки	Описание
			строки на диске, дополнительные байты добавляются перед значением этого типа, чтобы оно начиналось с указанной границы. Заданное выравнивание определяет смещение первого элемента последовательности. Возможные значения: • `c` = выравнивание по символам (`char`), то есть выравнивание не требуется. • `s` = выравнивание по коротким словам (`short`), 2 байта для большинства машин. • `i` = выравнивание по целым (`int`), 4 байта для большинства машин. • `d` = выравнивание по двойным словам (`double`), 8 байт для большинства машин, но не для всех. **Примечание** Для типов, используемых в системных таблицах, важно, чтобы размер и выравнивание, определённые в `pg_type` , согласовывались с тем, как компилятор располагает этот столбец в структуре,

Имя	Тип	Ссылки	Описание
			представляющей строку таблицы.
typstorage	char		Значение typstorage для типов varlena (типов с typlen = -1) говорит, готов ли тип для помещения в TOAST, и какова стратегия по умолчанию для атрибутов этого типа. Возможные значения: • p: Значение всегда должно храниться простым образом (plain). • e: Значение может храниться во «вторичном» отношении (если оно есть, см. pg_ class.reltoastrelid). • m: Значение может храниться сжатым внутри строки. • x: Значение может храниться сжатым внутри строки или во «вторичном» хранилище. Заметьте, что столбцы m тоже могут быть перемещены во вторичное хранилище, но только в качестве последней меры (столбцы e и x перемещаются в первую очередь).
typnotnull	bool		Поле typnotnull представляет ограничение «не NULL» для типа. Применяется только для доменов.
typbasetype	oid	pg_type .oid	Если это домен (см. typtype), то typbasetype указывает на тип, на котором он основан. Ноль, если это не домен.

Имя	Тип	Ссылки	Описание
typtypmod	int4		Домены используют typtypmod для записи модификатора (typmod), применяемого к их базовому типу (-1, если базовый тип не использует typmod). Если тип не является доменом, принимает значение -1.
typndims	int4		Значение typndims задаёт число размерностей массива для домена, определённого поверх массива (то есть когда typbasetype — тип массива). Для типов, отличных от доменов поверх типов массивов, принимает значение 0.
typcollation	oid	pg_collation .oid	Значение typcollation задаёт правило сортировки для типа. Если тип не является сортируемым, оно будет нулевым. У базового типа, поддерживающего правила сортировки, в этом поле будет DEFAULT_COLLATION_ OID . Домен на базе сортируемого типа может иметь другой OID правила сортировки, если оно было изменено для домена.
typdefaultbin	pg_node_tree		Если поле typdefaultbin не NULL, в нём содержится представление выражения по умолчанию для этого типа (совместимое с nodeToString() . Это поле используется только для доменов.
typdefault	text		Поле typdefault содержит NULL, если с типом не связано значение по

Имя	Тип	Ссылки	Описание
			умолчанию. Если typdefaultbin не NULL, typdefault должно содержать понятную человеку версию выражения значения по умолчанию, записанного в typdefaultbin. Если typdefaultbin содержит NULL, а typdefault нет, то в typdefault находится внешнее представление значения по умолчанию, которое можно передать функции преобразования ввода и получить константу.
typacl	aclitem[]		Права доступа; за подробностями обратитесь к описанию GRANT и REVOKE

В Таблице 49.63 перечисляются определённые в системе значения typcategory. Если этот список будет дополняться в будущем, в него будут добавляться тоже буквы ASCII в верхнем регистре. Все другие символы ASCII зарезервированы для категорий, определяемых пользователями.

Таблица 49.63. Коды typcategory

Код	Категория
A	Типы массивов
B	Логические типы
C	Составные типы
D	Типы даты/времени
E	Типы-перечисления
G	Геометрические типы
I	Типы, описывающие сетевые адреса
N	Числовые типы
P	Псевдотипы
R	Диапазонные типы
S	Строковые типы
T	Интервальные типы
U	Пользовательские типы
V	Типы битовых строк
X	Неизвестный тип (unknown)

49.63. pg_user_mapping

В каталоге pg_user_mapping хранятся сопоставления локальных пользователей с удалёнными. Обычные пользователи не имеют доступа к этому каталогу, они должны использовать представление pg_user_mappings.

Таблица 49.64. Столбцы pg_user_mapping

Имя	Тип	Ссылки	Описание
oid	oid		Идентификатор строки (скрытый атрибут; должен выбираться явно)
umuser	oid	pg_authid .oid	OID сопоставляемой локальной роли, либо 0, если сопоставление задаётся для всех
umserver	oid	pg_foreign_server .oid	OID стороннего сервера, содержащего это сопоставление
umoptions	text[]		Специальные параметры сопоставления пользователей, в виде строк «ключ=значение»

49.64. Системные представления

В дополнение к системным каталогам, в Postgres Pro есть набор встроенных представлений. Некоторые системные представления содержат в себе некоторые популярные запросы к системным каталогам, а другие дают доступ к внутреннему состоянию сервера.

Информационная схема (см. Главу 34) содержит другой набор представлений, пересекающихся по функциональности с системными представлениям. Так как информационная схема соответствует стандарту SQL, тогда как описанные здесь представления свойственны только для Postgres Pro, обычно лучше использовать информационную схему, если через неё можно получить всю требуемую информацию.

В Таблице 49.65 перечислены описываемые здесь системные представления. Подробное описание каждого представления следует далее. Есть также дополнительные представления, дающие доступ к результатам работы сборщика статистики; они перечисляются в Таблице 27.2.

Кроме явно отмеченных исключений, все описанные здесь представления доступны только для чтения.

Таблица 49.65. Системные представления

Имя представления	Предназначение
pg_available_extensions	доступные расширения
pg_available_extension_versions	доступные версии расширений
pg_config	параметры конфигурации времени компиляции
pg_cursors	открытые курсоры
pg_file_settings	сводка содержимого файла конфигурации
pg_group	группы пользователей баз данных

Имя представления	Предназначение
pg_hba_file_rules	сводка содержимого файла конфигурации аутентификации клиентов
pg_indexes	индексы
pg_locks	блокировки, установленные или ожидаемые в данный момент
pg_matviews	материализованные представления
pg_policies	policies
pg_prepared_statements	подготовленные операторы
pg_prepared_xacts	подготовленные транзакции
pg_publication_tables	публикации и связанные с ними таблицы
pg_replication_origin_status	информация об источниках репликации, включая данные прогресса репликации
pg_replication_slots	информация о слотах репликации
pg_roles	роли баз данных
pg_rules	правила
pg_seclabels	метки безопасности
pg_sequences	последовательности
pg_settings	значения параметров
pg_shadow	пользователи базы данных
pg_stats	статистика планировщика
pg_tables	таблицы
pg_timezone_abbrevs	аббревиатуры часовых поясов
pg_timezone_names	имена часовых поясов
pg_user	пользователи базы данных
pg_user_mappings	сопоставления пользователей
pg_views	представления

49.65. pg_available_extensions

В представлении pg_available_extensions перечисляются расширения, доступные для установки. Также обратите внимание на каталог pg_extension, в котором перечисляются уже установленные расширения.

Таблица 49.66. Столбцы pg_available_extensions

Имя	Тип	Описание
name	name	Имя расширения
default_version	text	Имя версии по умолчанию, либо NULL, если это имя не определено
installed_version	text	Текущая установленная версия расширения, либо NULL, если расширение не установлено
comment	text	Строка комментария из управляющего файла расширения

Представление pg_available_extensions доступно только для чтения.

49.66. pg_available_extension_versions

В представлении pg_available_extension_versions перечислены определённые версии расширений, доступные для установки. Также обратите внимание на каталог pg_extension, в котором перечисляются уже установленные расширения.

Таблица 49.67. Столбцы pg_available_extension_versions

Имя	Тип	Описание
name	name	Имя расширения
version	text	Имя версии
installed	bool	True, если эта версия расширения уже установлена
superuser	bool	True, если устанавливать это расширение разрешено только суперпользователям
relocatable	bool	True, если расширение можно переместить в другую схему
schema	name	Имя схемы, в которую должно быть установлено расширение, либо NULL, если оно частично или полностью переместимо
requires	name[]	Имена расширений, требующихся для данного, либо NULL, если таких нет
comment	text	Строка комментария из управляющего файла расширения

Представление pg_available_extension_versions доступно только для чтения.

49.67. pg_config

В представлении pg_config описываются конфигурационные параметры времени компиляции для текущей установленной версии Postgres Pro. Оно предназначено, например, для программных средств, которым может потребоваться узнать у Postgres Pro расположение требуемых заголовочных файлов и библиотек. Оно выдаёт ту же базовую информацию, что и клиентская утилита Postgres Pro pg_config.

По умолчанию представление pg_config доступно только суперпользователям и только для чтения.

Таблица 49.68. Столбцы pg_config

Имя	Тип	Описание
name	text	Имя параметра
setting	text	Значение параметра

49.68. pg_cursors

В представлении pg_cursors перечисляются курсоры, доступные в данный момент. Курсоры могут быть созданы несколькими способами:

- через оператор DECLARE в SQL
- через сообщение Bind в клиент-серверном протоколе, как описано в Подразделе 50.2.3

- через интерфейс программирования сервера (SPI, Server Programming Interface), как описано в Разделе 44.1

В представлении pg_cursors показываются курсоры, полученные любым способом. Курсор существует только на протяжении транзакции, в которой он определён, если только он не был объявлен с указанием WITH HOLD. Поэтому не удерживаемые курсоры показываются в этом представлении только пока не завершится создавшая их транзакция.

> **Примечание**
>
> Курсоры используются внутри системы для реализации некоторых компонентов Postgres Pro, таких как процедурные языки. Таким образом, в представлении pg_cursors могут быть курсоры, которые пользователи не создавали явно.

Таблица 49.69. Столбцы pg_cursors

Имя	Тип	Описание
name	text	Имя курсора
statement	text	Дословная строка запроса, создавшего данный курсор
is_holdable	boolean	True, если курсор удерживаемый (то есть, к нему можно обращаться после того, как будет зафиксирована транзакция, в которой он объявлен); в противном случае — false
is_binary	boolean	True, если курсор был объявлен с указанием BINARY; в противном случае — false
is_scrollable	boolean	True, если курсор прокручиваемый (то есть, позволяет получать строки непоследовательным образом); в противном случае — false
creation_time	timestamptz	Время, в которое был объявлен курсор

Представление pg_cursors доступно только для чтения.

49.69. pg_file_settings

В представлении pg_file_settings показывается сводное содержимое файлов конфигурации сервера. Для каждой имеющейся в этих файлах записи «имя = значение» это представление содержит строку с отметкой, показывающей, может ли это значение быть успешно применено. Также это представление может содержать дополнительные строки, говорящие о проблемах, не связанных с записями «имя = значение», например, синтаксических ошибках в этих файлах.

Это представление полезно для проверки, будут ли работать планируемые изменения в файлах конфигурации, или для диагностики возникшей ранее проблемы. Заметьте, что в этом представлении отражается *текущее* содержимое файлов, а не то, что было применено сервером в последний раз. (Чтобы получить то состояние, обычно достаточно обратиться к представлению pg_settings.)

По умолчанию представление pg_file_settings доступно только суперпользователям и только для чтения.

Таблица 49.70. Столбцы pg_file_settings

Имя	Тип	Описание
sourcefile	text	Полный путь и имя файла конфигурации
sourceline	integer	Номер строки в файле конфигурации, из которой получена эта запись
seqno	integer	Порядок, в котором обрабатываются записи (1..*n*)
name	text	Имя параметра конфигурации
setting	text	Значение, присваиваемое параметру
applied	boolean	True, если значение может быть применено успешно
error	text	Сообщение об ошибке, говорящее, почему эта запись не может быть применена, либо NULL

Если файл конфигурации содержит синтаксические ошибки или недопустимые имена параметров, сервер не будет пытаться применять никакие параметры из него, так что все поля applied будут равны False. В этом случае представление будет содержать одну или несколько строк, в которых поле error описывает проблему. Иначе отдельные записи этого файла будут применяться по возможности. Если заданное в некоторой записи присваивание выполнить нельзя (например, из-за неверного значения или если параметр нельзя изменять после запуска сервера), в поле error для неё будет записано соответствующее сообщение. Поле applied также может содержать False, если данная запись переопределяется последующей записью с тем же именем параметра; это не считается ошибкой, так что поле error будет пустым.

Чтобы узнать больше о различных способах изменения параметров времени выполнения, обратитесь к Разделу 18.1.

49.70. pg_group

Представление pg_group существует для обратной совместимости: оно эмулирует каталог, существовавший в PostgreSQL до версии 8.1. В нём показываются имена и члены всех ролей без признака rolcanlogin, что приблизительно соответствует списку ролей, которые использовались как группы.

Таблица 49.71. Столбцы pg_group

Имя	Тип	Ссылки	Описание
groname	name	pg_authid .rolname	Имя группы
grosysid	oid	pg_authid .oid	ID этой группы
grolist	oid[]	pg_authid .oid	Массив, содержащий идентификаторы ролей в этой группе

49.71. pg_hba_file_rules

В представлении pg_hba_file_rules показывается сводное содержимое файла конфигурации аутентификации клиентов, pg_hba.conf. Для каждой непустой и незакомментированной строки в этом файле данное представление содержит одну строку с отметкой, показывающей, может ли это правило быть успешно применено.

Это представление может быть полезно для проверки, будут ли работать планируемые изменения в файле конфигурации аутентификации, или для диагностики возникшей проблемы. Заметьте, что в этом представлении отражается *текущее* содержимое файла, а не то, что было загружено сервером в последний раз.

По умолчанию представление `pg_hba_file_rules` доступно только суперпользователям и только для чтения.

Таблица 49.72. Столбцы `pg_hba_file_rules`

Имя	Тип	Описание
`line_number`	`integer`	Номер строки этого правила в `pg_hba.conf`
`type`	`text`	Тип подключения
`database`	`text[]`	Список имён баз данных, к которым применяется это правило
`user_name`	`text[]`	Список имён пользователей и групп, к которым применяется это правило
`address`	`text`	Имя или IP-адрес узла либо одно из значений: `all`, `samehost` или `samenet`, либо NULL для локальных подключений
`netmask`	`text`	Маска IP-адреса либо NULL, если это неприменимо
`auth_method`	`text`	Метод аутентификации
`options`	`text[]`	Параметры, задаваемые для метода аутентификации (если они есть)
`error`	`text`	Сообщение об ошибке, говорящее, почему эта строка не может быть обработана, либо NULL

Обычно строка, отражающая некорректную запись, будет содержать значения только в полях `line_number` и `error`.

Чтобы узнать больше о конфигурации аутентификации клиентов, обратитесь к Главе 19.

49.72. `pg_indexes`

Представление `pg_indexes` даёт доступ к полезной информации обо всех индексах в базе данных.

Таблица 49.73. Столбцы `pg_indexes`

Имя	Тип	Ссылки	Описание
`schemaname`	`name`	`pg_namespace .nspname`	Имя схемы, содержащей таблицу и индекс
`tablename`	`name`	`pg_class .relname`	Имя таблицы, для которой создан индекс
`indexname`	`name`	`pg_class .relname`	Имя индекса
`tablespace`	`name`	`pg_tablespace .spcname`	Имя табличного пространства,

Имя	Тип	Ссылки	Описание
			содержащего индекс (NULL, если это пространство по умолчанию)
indexdef	text		Определение индекса (реконструированная команда CREATE INDEX)

49.73. pg_locks

Представление pg_locks даёт доступ к информации о блокировках, удерживаемых активными процессами на сервере баз данных. Подробнее блокировки рассматриваются в Главе 13.

Представление pg_locks содержит одну строку для каждого активного блокируемого объекта, запрошенного режима блокировки и блокирующего процесса. Таким образом, один и тот же блокируемый объект может фигурировать в этом представлении неоднократно, если его блокируют или ожидают блокировки несколько процессов. Однако объекты, свободные от блокировок, в этом представлении отсутствуют вовсе.

Существует несколько различных типов блокируемых объектов: отношения целиком (например, таблицы), отдельные страницы отношений, отдельные кортежи отношений, идентификаторы транзакций (виртуальные и постоянные) и произвольные объекты баз данных (идентифицируемые по OID класса и OID объекта, так же как в pg_description или pg_depend). Кроме того, в виде отдельного блокируемого объекта представлено право расширения отношения, как и право изменения значения pg_database.datfrozenxid. Также могут быть установлены «рекомендательные» блокировки, не имеющие предопределённого значения.

Таблица 49.74. Столбцы pg_locks

Имя	Тип	Ссылки	Описание
locktype	text		Тип блокируемого объекта: relation (отношение), extend (расширение отношения), frozenid (замороженный идентификатор), page (страница), tuple (кортеж), transactionid (идентификатор транзакции), virtualxid (виртуальный идентификатор), object (объект), userlock (пользовательская блокировка) или advisory (рекомендательная)
database	oid	pg_database .oid	OID базы данных, к которой относится цель блокировки, ноль, если это разделяемый объект, либо NULL,

Имя	Тип	Ссылки	Описание
			если целью является идентификатор транзакции
relation	oid	pg_class .oid	OID отношения, являющегося целью блокировки, либо NULL, если цель блокировки — не отношение или часть отношения
page	integer		Номер страницы в отношении, являющейся целью блокировки, либо NULL, если цель блокировки — не страница или кортеж отношения
tuple	smallint		Номер кортежа на странице, являющегося целью блокировки, либо NULL, если цель блокировки — не кортеж
virtualxid	text		Виртуальный идентификатор транзакции, являющийся целью блокировки, либо NULL, если цель блокировки — другой объект
transactionid	xid		Идентификатор транзакции, являющийся целью блокировки, либо NULL, если цель блокировки — другой объект
classid	oid	pg_class .oid	OID системного каталога, содержащего цель блокировки, либо NULL, если цель блокировки — не обычный объект базы данных
objid	oid	любой столбец OID	OID цели блокировки в соответствующем системном каталоге, либо NULL, если цель блокировки — не обычный объект базы данных

Имя	Тип	Ссылки	Описание
objsubid	smallint		Номер столбца, являющегося целью блокировки (на саму таблицу указывают classid и objid), ноль, если это некоторый другой обычный объект базы данных, либо NULL, если цель не обычный объект
virtualtransaction	text		Виртуальный идентификатор транзакции, удерживающей или ожидающей блокировку
pid	integer		Идентификатор серверного процесса (PID, Process ID), удерживающего или ожидающего эту блокировку, либо NULL, если блокировка удерживается подготовленной транзакцией
mode	text		Название режима блокировки, которая удерживается или запрашивается этим процессом (см. Подраздел 13.3.1 и Подраздел 13.2.3)
granted	boolean		True, если блокировка получена, и false, если она ожидается
fastpath	boolean		True, если блокировка получена по короткому пути, и false, если она получена через основную таблицу блокировок

Признак granted устанавливается в строке, представляющей блокировку, удерживаемую указанным процессом. Если он сброшен, этот процесс ждёт блокировки, из чего следует что как минимум один другой процесс удерживает или ожидает блокировку того же объекта в конфликтующем режиме. Ожидающий процесс будет приостановлен до освобождения другой блокировки (или выявления ситуации взаимоблокировки). Один процесс в один момент времени может ожидать получения максимум одной блокировки.

На протяжении транзакции серверный процесс удерживает исключительную блокировку виртуального идентификатора транзакции. Если транзакции назначается постоянный идентификатор (что обычно происходит, только если транзакция изменяет состояние базы данных), он также удерживает до её завершения блокировку этого постоянного идентификатора. Когда процесс находит необходимым ожидать именно какую-то другую транзакцию, он делает

это, запрашивая разделяемую блокировку для идентификатора этой транзакции (виртуального или постоянного, в зависимости от ситуации). Этот запрос будет выполнен, только когда другая транзакция завершится и освободит свои блокировки.

Хотя кортежи тоже представляют собой блокируемый объект, информация о блокировках строк хранится на диске, а не в памяти, поэтому такие блокировки обычно не показываются в этом представлении. Если процесс ожидает блокировки на уровне строки, он обычно виден в нём как ожидающий постоянного идентификатора транзакции текущего владельца этой блокировки.

Рекомендательные блокировки могут быть получены по ключам, состоящим из одного значения `bigint` или из двух значений integer. Старшая половина `bigint` выводится в столбце `classid`, а младшая половина в столбце `objid`, и `objsubid` равен 1. Исходное значение `bigint` может быть восстановлено выражением `(classid::bigint << 32) | objid::bigint`. Для ключей integer первая часть ключа находится в `classid`, а вторая часть в `objid`, и `objsubid` равна 2. Конкретное предназначение этих ключей определяет пользователь. Рекомендательные блокировки существуют в рамках базы данных, поэтому столбец `database` имеет значение для таких блокировок.

Представление `pg_locks` даёт общую информацию по всем блокировкам в кластере баз данных, а не только по тем, что относятся к текущей базе. Хотя соединив `relation` с `pg_class.oid`, можно получить заблокированные отношения, это будет работать корректно только для отношений в текущей базе данных (для тех, в блокировках которых столбец `database` содержит OID текущей базы данных или ноль).

Соединив столбец `pid` со столбцом `pid` представления `pg_stat_activity`, можно получить дополнительную информацию о сеансах, удерживающих или ожидающих каждую блокировку, например так:

```
SELECT * FROM pg_locks pl LEFT JOIN pg_stat_activity psa
    ON pl.pid = psa.pid;
```

Также, если вы используете подготовленные транзакции, столбец `virtualtransaction` можно соединить со столбцом `transaction` представления `pg_prepared_xacts` для получения дополнительной информации о подготовленных транзакциях, удерживающих блокировки. (Подготовленная транзакция не может ожидать блокировок, но она может продолжать удерживать блокировки, полученные ей в процессе выполнения.) Например:

```
SELECT * FROM pg_locks pl LEFT JOIN pg_prepared_xacts ppx
    ON pl.virtualtransaction = '-1/' || ppx.transaction;
```

Хотя в принципе возможно получить информацию о процессах, которые блокируют другие процессы, соединив представление `pg_locks` с ним же, очень трудно сделать это правильно во всех деталях. В частности потому, что такой запрос должен будет знать, какие режимы блокировки конфликтуют с другими. Мало того, представление `pg_locks` не показывает, какие процессы стоят перед какими в очередях ожидания блокировок, а также какие процессы являются параллельными рабочими процессами и к каким клиентским сеансам они относятся. Чтобы узнать, каким процессом или процессами блокируется ожидающий процесс, лучше использовать функцию `pg_blocking_pids()` (см. Таблицу 9.60).

В представлении `pg_locks` показываются данные и из менеджера обычных блокировок, и из менеджера предикатных блокировок, которые являются отдельными механизмами; кроме того, менеджер обычных блокировок подразделяет свои блокировки на обычные и полученные *быстрым путём*. Абсолютная согласованность всех этих данных не гарантируется. При обращении к этому представлению данные блокировок по быстрому пути (с `fastpath = true`) собираются по очереди с каждого серверного процесса, без замораживания состояния всего менеджера блокировок, так что существует возможность, что в процессе сбора этой информации блокировки будут освобождены или получены. Заметьте однако, что эти блокировки не должны конфликтовать с любыми другими актуальными блокировками. После того как от всех процессов получены блокировки по быстрому пути, менеджер обычных блокировок замораживается целиком

и информация обо всех оставшихся блокировках собирается в атомарной операции. После размораживания этого менеджера, также замораживается менеджер предикатных блокировок, и информация об этих блокировках собирается атомарно. Таким образом, за исключением блокировок по быстрому пути, каждый менеджер блокировок выдаёт согласованный набор результатов, но так как мы не блокируем оба этих менеджера одновременно, блокировки могут быть получены или освобождены после того, как опрашивается менеджер обычных блокировок, и до того, как опрашивается менеджер предикатных блокировок.

Блокировка менеджера обычных или предикатных блокировок может отразиться на производительности базы данных, если обращаться к этому представлению часто. Эта блокировка удерживается не дольше, чем необходимо для получения данных от менеджеров, но это не исключает возможность снижения производительности.

49.74. pg_matviews

Представление pg_matviews даёт доступ к полезной информации обо всех материализованных представлениях в базе данных.

Таблица 49.75. Столбцы pg_matviews

Имя	Тип	Ссылки	Описание
schemaname	name	pg_namespace .nspname	Имя схемы, содержащей материализованное представление
matviewname	name	pg_class .relname	Имя материализованного представления
matviewowner	name	pg_authid .rolname	Имя владельца материализованного представления
tablespace	name	pg_tablespace .spcname	Имя табличного пространства, содержащего материализованное представление (NULL, если это пространство по умолчанию)
hasindexes	boolean		True, если материализованное представление имеет (или недавно имело) индексы
ispopulated	boolean		True, если материализованное представление в данный момент наполнено
definition	text		Определение материализованного представления (реконструированный запрос SELECT)

49.75. pg_policies

Представление pg_policies даёт доступ к полезной информации обо всех политиках защиты на уровне строк в базе данных.

Таблица 49.76. Столбцы pg_policies

Имя	Тип	Ссылки	Описание
schemaname	name	pg_namespace .nspname	Имя схемы, содержащей таблицу с этой политикой
tablename	name	pg_class .relname	Имя таблицы с этой политикой
policyname	name	pg_policy .polname	Имя политики
polpermissive	text		Является ли политика разрешительной или ограничительной?
roles	name[]		Роли, к которым применяется политика
cmd	text		Тип команды, к которому применяется политика
qual	text		Выражение, добавляемое к условиям барьера безопасности в запросы, к которым применяется политика
with_check	text		Выражение, добавляемое к условиям WITH CHECK в запросы, которые пытаются добавлять строки в таблицу

49.76. pg_prepared_statements

В представлении pg_prepared_statements отображаются все подготовленные операторы, существующие в текущем сеансе. За дополнительными сведениями о подготовленных операторах обратитесь к PREPARE.

Представление pg_prepared_statements содержит отдельную строку для каждого подготовленного оператора. Строки добавляются в него, когда создаётся новый подготовленный оператор, и удаляются, когда подготовленный оператор освобождается (например, командой DEALLOCATE).

Таблица 49.77. Столбцы pg_prepared_statements

Имя	Тип	Описание
name	text	Идентификатор подготовленного оператора
statement	text	Строка запроса, переданного клиентом и создавшего этот подготовленный оператор. Для подготовленных операторов, создаваемых через SQL, это оператор PREPARE, переданный клиентом. Для подготовленных

Имя	Тип	Описание
		операторов, созданных через клиент-серверный протокол, этот текст представляет сам подготовленный оператор.
prepare_time	timestamptz	Время, в которое был создан подготовленный оператор
parameter_types	regtype[]	Ожидаемые типы параметров для подготовленного оператора в форме массива regtype. OID, соответствующий элементу этого массива, может быть получен в результате приведения значения regtype к oid.
from_sql	boolean	Значение true, если подготовленный оператор был создан SQL-командой PREPARE; false, если оператор был подготовлен через клиент-серверный протокол

Представление pg_prepared_statements доступно только для чтения.

49.77. pg_prepared_xacts

Представление pg_prepared_xacts содержит информацию о транзакциях, которые в настоящее время подготовлены для двухфазной фиксации (за подробностями обратитесь к PREPARE TRANSACTION).

Представление pg_prepared_xacts содержит отдельную запись для каждой подготовленной транзакции. Эта запись удаляется, когда транзакция фиксируется или откатывается.

Таблица 49.78. Столбцы pg_prepared_xacts

Имя	Тип	Ссылки	Описание
transaction	xid		Числовой идентификатор подготовленной транзакции
gid	text		Глобальный идентификатор транзакции, назначаемый транзакции
prepared	timestamp with time zone		Время, в которое транзакция была подготовлена для фиксации
owner	name	pg_authid .rolname	Имя пользователя, выполнявшего транзакцию
database	name	pg_database .datname	Имя базы данных, в которой выполнялась транзакция

Когда запрашивается представление pg_prepared_xacts, внутренние структуры данных менеджера транзакций на мгновение блокируются и создаётся их копия для вывода через это представление. Это гарантирует, что представление выдаёт согласованный набор результатов, при этом не задерживая обычные операции на более продолжительное время, чем необходимо. Тем не менее это может отрицательно сказаться на производительности базы данных при частых обращениях к представлению.

49.78. pg_publication_tables

Представление pg_publication_tables содержит информацию о сопоставлениях публикаций с таблицами, которые в них содержатся. В отличие от нижележащего каталога pg_publication_rel, в это представление добавляются публикации, определённые как FOR ALL TABLES, так что с такими публикациями будет присутствовать строка для каждой соответствующей таблицы.

Таблица 49.79. Столбцы pg_publication_tables

Имя	Тип	Ссылки	Описание
pubname	name	pg_publication.pubname	Имя публикации
schemaname	name	pg_namespace.nspname	Имя схемы, содержащей таблицу
tablename	name	pg_class.relname	Имя таблицы

49.79. pg_replication_origin_status

Представление pg_replication_origin_status содержит информацию о позиции воспроизведения записей репликации, достигнутой для определённого источника. Подробно источники репликации описаны в Главе 47.

Таблица 49.80. Столбцы pg_replication_origin_status

Имя	Тип	Ссылки	Описание
local_id	Oid	pg_replication_origin.roident	внутренний идентификатор узла
external_id	text	pg_replication_origin.roname	внешний идентификатор узла
remote_lsn	pg_lsn		LSN исходного узла, до которого были реплицированы данные.
local_lsn	pg_lsn		LSN данного узла, с которым был реплицирован remote_lsn. Применяется при асинхронной фиксации для сброса на диск записей фиксации до сохранения данных.

49.80. pg_replication_slots

Представление pg_replication_slots содержит список всех слотов репликации, существующих в данный момент в кластере баз данных, а также их текущее состояние.

За дополнительной информацией о слотах репликации обратитесь к Подразделу 25.2.6 и Главе 46.

Таблица 49.81. Столбцы `pg_replication_slots`

Имя	Тип	Ссылки	Описание
`slot_name`	`name`		Уникальный в рамках кластера идентификатор для слота репликации
`plugin`	`name`		Базовое имя разделяемого объекта, содержащего модуль вывода, который используется этим логическим слотом, либо NULL для физических слотов.
`slot_type`	`text`		Тип слота — `physical` (физический) или `logical` (логический)
`datoid`	`oid`	`pg_database` .oid	OID базы данных, с которой связан этот слот, либо NULL. С базой данных могут быть связаны только логические слоты.
`database`	`text`	`pg_database` .datname	Имя базы данных, с которой связан этот слот, либо NULL. С базой данных могут быть связаны только логические слоты.
`temporary`	`boolean`		True, если это временный слот репликации. Временные слоты не сохраняются на диск и автоматически удаляются при ошибке или завершении сеанса.
`active`	`boolean`		True, если слот активно используется в данный момент
`active_pid`	`integer`		ID процесса сеанса, занимающего этот слот, если данный слот активно используется в данный момент. `NULL`, если он не используется.
`xmin`	`xid`		Старейшая транзакция, которая должна сохраняться в базе данных для этого слота. `VACUUM` не сможет удалять

Имя	Тип	Ссылки	Описание
			кортежи, удалённые более поздними транзакциями.
catalog_xmin	xid		Старейшая транзакция, затрагивающая системные каталоги, которая должна сохраняться в базе данных для этого слота. VACUUM не сможет удалять кортежи, удалённые более поздними транзакциями.
restart_lsn	pg_lsn		Адрес (LSN) старейшей записи в WAL, которая по-прежнему может быть нужна пользователям этого слота и, таким образом, не будет автоматически удаляться при контрольных точках.
confirmed_flush_lsn	pg_lsn		Адрес (LSN), до которого потребитель логического слота подтвердил получение данных. Данные старее этого LSN уже недоступны. Для физических слотов — NULL.

49.81. pg_roles

Представление pg_roles открывает доступ к информации о ролях в базах данных. Это просто доступное для всех отображение каталога pg_authid, в котором очищено поле пароля.

В этом представлении выводится OID из нижележащей таблицы, так как это необходимо для соединений с другими каталогами.

Таблица 49.82. Столбцы pg_roles

Имя	Тип	Ссылки	Описание
rolname	name		Имя роли
rolsuper	bool		Роли имеет права суперпользователя
rolinherit	bool		Роль автоматически наследует права ролей, в которые она включена

Имя	Тип	Ссылки	Описание
rolcreaterole	bool		Роль может создавать другие роли
rolcreatedb	bool		Роль может создавать базы данных
rolcanlogin	bool		Роль может подключаться к серверу. То есть эта роль может быть указана в качестве начального идентификатора авторизации сеанса
rolreplication	bool		Роль является ролью репликации. Такие роли могут устанавливать соединения для репликации и создавать/удалять слоты репликации.
rolconnlimit	int4		Для ролей, которые могут подключаться к серверу, это значение задаёт максимально разрешённое для этой роли число одновременных подключений. При значении -1 ограничения нет.
rolpassword	text		Не пароль (всегда выводится как ********)
rolvaliduntil	timestamptz		Срок действия пароля (используется только при аутентификации по паролю); NULL, если срок действия не ограничен
rolbypassrls	bool		Роль не подчиняется никаким политикам защиты на уровне строк; за подробностями обратитесь к Разделу 5.7.
rolconfig	text[]		Заданные для роли значения по умолчанию переменных времени конфигурации
oid	oid	pg_authid .oid	Идентификатор роли

49.82. pg_rules

Представление pg_rules открывает доступ к полезной информации о правилах перезаписи запросов.

Таблица 49.83. Столбцы pg_rules

Имя	Тип	Ссылки	Описание
schemaname	name	pg_namespace .nspname	Имя схемы, содержащей таблицу
tablename	name	pg_class .relname	Имя таблицы, с которой связано это правило
rulename	name	pg_rewrite .rulename	Имя правила
definition	text		Определение правила (реконструированная команда создания)

Из представления pg_rules исключены правила ON SELECT для представлений и материализованных представлений; их можно увидеть в pg_views и pg_matviews.

49.83. pg_seclabels

Представление pg_seclabels содержит информацию о метках безопасности. Это более простая в использовании версия каталога pg_seclabel.

Таблица 49.84. Столбцы pg_seclabels

Имя	Тип	Ссылки	Описание
objoid	oid	любой столбец OID	OID объекта, к которому относится эта метка безопасности
classoid	oid	pg_class .oid	OID системного каталога, к которому относится этот объект
objsubid	int4		Для метки безопасности, связанной со столбцом таблицы, это номер столбца (objoid и classoid указывают на саму таблицу). Для всех других типов объектов это поле содержит ноль.
objtype	text		Тип объекта, к которому применяется эта метка, в текстовом виде.
objnamespace	oid	pg_namespace .oid	OID пространства имён объекта, если применимо; иначе NULL.
objname	text		Имя объекта, к которому применяется эта метка, в текстовом виде.

Имя	Тип	Ссылки	Описание
provider	text	pg_seclabel .provider	Поставщик меток безопасности, связанный с этой меткой.
label	text	pg_seclabel .label	Метка безопасности, применённая к этому объекту.

49.84. pg_sequences

Представление pg_sequences даёт доступ к полезной информации обо всех последовательностях в базе данных.

Таблица 49.85. Столбцы pg_sequences

Имя	Тип	Ссылки	Описание
schemaname	name	pg_namespace .nspname	Имя схемы, содержащей последовательность
sequencename	name	pg_class .relname	Имя последовательности
sequenceowner	name	pg_authid .rolname	Имя владельца последовательности
data_type	regtype	pg_type .oid	Тип данных последовательности
start_value	bigint		Начальное значение последовательности
min_value	bigint		Минимальное значение последовательности
max_value	bigint		Максимальное значение последовательности
increment_by	bigint		Шаг увеличения последовательности
cycle	boolean		Зацикливается ли последовательность
cache_size	bigint		Размер кеша последовательности
last_value	bigint		Последнее значение последовательности, записанное на диск. Если используется кеширование, это значение может быть больше последнего значения, полученного из последовательности. Если к этой последовательности ещё не обращались, содержит NULL. Также содержит NULL, если

Имя	Тип	Ссылки	Описание
			текущий пользователь не имеет права USAGE или SELECT.

49.85. pg_settings

Представление pg_settings открывает доступ к параметрам времени выполнения сервера. По сути оно представляет собой альтернативный интерфейс для команд SHOW и SET. Оно также позволяет получить некоторые свойства каждого параметра, которые нельзя получить непосредственно, используя команду SHOW, например, минимальные и максимальные значения.

Таблица 49.86. Столбцы pg_settings

Имя	Тип	Описание
name	text	Имя параметра конфигурации времени выполнения
setting	text	Текущее значение параметра
unit	text	Неявно подразумеваемая единица измерения параметра
category	text	Логическая группа параметра
short_desc	text	Краткое описание параметра
extra_desc	text	Дополнительное, более подробное, описание параметра
context	text	Контекст, в котором может задаваться значение параметра (см. ниже)
vartype	text	Тип параметра (bool, enum, integer, real или string)
source	text	Источник текущего значения параметра
min_val	text	Минимальное допустимое значение параметра (NULL для нечисловых значений)
max_val	text	Максимально допустимое значение параметра (NULL для нечисловых значений)
enumvals	text[]	Допустимые значения параметра-перечисления (NULL для значений не перечислений)
boot_val	text	Значение параметра, устанавливаемое при запуске сервера, если параметр не устанавливается другим образом
reset_val	text	Значение, к которому будет сбрасывать параметр команда RESET в текущем сеансе
sourcefile	text	Файл конфигурации, в котором было задано текущее значение (NULL для значений,

Имя	Тип	Описание
		полученных не из файлов конфигурации, или при чтении этого поля пользователем, который не является суперпользователем или членом роли pg_read_all_settings); полезно при использовании указаний include в файлах конфигурации
sourceline	integer	Номер строки в файле конфигурации, в которой было задано текущее значение (NULL для значений, полученных не из файлов конфигурации, или при чтении этого поля не суперпользователем и не членом роли pg_read_all_settings).
pending_restart	boolean	true, если значение изменено в файле конфигурации, но требуется перезапуск; в противном случае — false.

Поле context может содержать одно из следующих значений (они перечислены в порядке уменьшения сложности изменения параметров):

internal

Эти параметры нельзя изменить непосредственно; они отражают значения, определяемые внутри системы. Некоторые из них можно изменить, пересобрав сервер с другими параметрами конфигурации, либо передав другие аргументы команде initdb.

postmaster

Эти параметры могут быть применены только при запуске сервера, так что любое изменение требует перезапуска сервера. Значения этих параметров обычно задаются в postgresql.conf, либо передаются в командной строке при запуске сервера. Разумеется, параметры более низкого уровня context также можно задать в момент запуска сервера.

sighup

Внесённые в postgresql.conf изменения этих параметров можно применить, не перезапуская сервер. Если передать управляющему процессу сигнал SIGHUP, он перечитает postgresql.conf и применит изменения. Управляющий процесс также перешлёт сигнал SIGHUP всем своим дочерним процессам, чтобы они тоже приняли новое значение.

superuser-backend

Внесённые в postgresql.conf изменения этих параметров можно применить без перезапуска сервера; их также можно задать для определённого сеанса в пакете запроса соединения (например, через переменную окружения PGOPTIONS, учитываемую библиотекой libpq), но сделать это может только суперпользователь. Однако эти параметры никогда не меняются в сеансе, когда он уже начат. Если вы измените их в postgresql.conf, отправьте сигнал SIGHUP управляющему процессу, чтобы он перечитал postgresql.conf. Новые значения подействуют только на сеансы, запускаемые после этого.

backend

Внесённые в postgresql.conf изменения этих параметров можно применить без перезапуска сервера; их также можно задать для определённого сеанса в пакете запроса соединения (например, через переменную окружения PGOPTIONS, учитываемую библиотекой libpq); это может сделать любой пользователь в своём сеансе. Однако эти параметры никогда не меняются в сеансе, когда он уже начат. Если вы измените их в postgresql.conf, отправьте сигнал SIGHUP управляющему процессу, чтобы он перечитал postgresql.conf. Новые значения подействуют только на сеансы, запускаемые после этого.

superuser

Эти параметры можно изменить в postgresql.conf, либо в рамках сеанса, командой SET; но только суперпользователи могут менять их, используя SET. Изменения в postgresql.conf будут отражены в существующих сеансах, только если в них командой SET не были заданы локальные значения.

user

Эти параметры можно задать в postgresql.conf, либо в рамках сеанса, командой SET. В рамках сеанса изменять их разрешено всем пользователям. Изменения в postgresql.conf будут отражены в существующих сеансах, только если в них командой SET не были заданы локальные значения.

Чтобы узнать больше о различных способах изменения этих параметров, обратитесь к Разделу 18.1.

Представление pg_settings не допускает добавление и удаление строк, но допускает изменение. Команда UPDATE, применённая к строке pg_settings, равнозначна выполнению команды SET для этого параметра. Изменение повлияет только на значение в текущем сеансе. Если UPDATE выполняется в транзакции, которая затем прерывается, эффект UPDATE пропадает, когда транзакция откатывается. После фиксирования окружающей транзакции этот эффект сохраняется до завершения сеанса, если он не будет переопределён другой командой UPDATE или SET.

49.86. pg_shadow

Представление pg_shadow существует для обратной совместимости: оно эмулирует каталог, существовавший в PostgreSQL до версии 8.1. В нём показываются свойства всех ролей с признаком rolcanlogin в pg_authid.

Такое имя («тень») объясняется тем фактом, что эта таблица не должна быть доступна на чтение всем, так как она содержит пароли. Представление pg_user является доступным всем отображением pg_shadow, в котором очищено поле пароля.

Таблица 49.87. Столбцы pg_shadow

Имя	Тип	Ссылки	Описание
usename	name	pg_authid .rolname	Имя пользователя
usesysid	oid	pg_authid .oid	ID этого пользователя
usecreatedb	bool		Пользователь может создавать базы данных
usesuper	bool		Пользователь является суперпользователем
userepl	bool		Пользователь может инициировать потоковую репликацию, включать и отключать режим резервного копирования.

Имя	Тип	Ссылки	Описание
usebypassrls	bool		Пользователь не подчиняется никаким политикам защиты на уровне строк; за подробностями обратитесь к Разделу 5.7.
passwd	text		Пароль (возможно зашифрованный); NULL, если он не задан. Подробнее хранение зашифрованных паролей описано в pg_authid .
valuntil	abstime		Срок действия пароля (используется только при аутентификации по паролю)
useconfig	text[]		Сеансовые значения по умолчанию для переменных конфигурации времени выполнения

49.87. pg_stats

Представление pg_stats открывает доступ к информации, хранящейся в каталоге pg_statistic. Это представление даёт доступ только к тем строкам каталога pg_statistic, что соответствуют таблицам, которые пользователь может читать; таким образом, это представление можно без опасений разрешить читать всем.

Кроме того, представление pg_stats специально разработано для подачи информации в более понятном виде, чем нижележащий каталог — ценой того, что схему представления приходится расширять всякий раз, когда для pg_statistic определяются новые типы слотов.

Таблица 49.88. Столбцы pg_stats

Имя	Тип	Ссылки	Описание
schemaname	name	pg_namespace .nspname	Имя схемы, содержащей таблицу
tablename	name	pg_class .relname	Имя таблицы
attname	name	pg_attribute .attname	Имя столбца, описываемого этой строкой
inherited	bool		Если true, в данных этой строки учитываются значения в дочерних столбцах, а не только в указанной таблице
null_frac	real		Доля записей, в которых этот столбец содержит NULL

Имя	Тип	Ссылки	Описание
avg_width	integer		Средний размер элементов в столбце, в байтах
n_distinct	real		Число больше нуля представляет примерное количество различных значений в столбце. Если это число меньше нуля, его модуль представляет количество различных значений, делённое на количество строк. (Отрицательная форма применяется, когда ANALYZE полагает, что число различных значений, скорее всего, будет расти по мере роста таблицы; положительная, когда в столбце, вероятно, будет фиксированное количество возможных значений.) Например, -1 указывает на столбец с уникальным содержимым, в котором количество различных значений совпадает с количеством строк.
most_common_vals	anyarray		Список самых частых значений в столбце. (NULL, если не находятся значения, встречающиеся чаще других.)
most_common_freqs	real[]		Список частот самых частых значений, то есть число их вхождений, делённое на общее количество строк. (NULL, когда most_common_vals — NULL.)
histogram_bounds	anyarray		Список значений, разделяющих значения столбца на примерно одинаковые популяции. Значения most_common_vals , если они присутствуют, не рассматриваются при

Имя	Тип	Ссылки	Описание
			вычислении этой гистограммы. (Этот столбец содержит NULL, если для типа данных столбца не определён оператор <, либо если в most_common_vals перечисляется вся популяция.)
correlation	real		Статистическая корреляция между физическим порядком строк и логическим порядком значений столбца. Допустимые значения лежат в диапазоне -1 .. +1. Когда значение около -1 или +1, сканирование индекса по столбцу будет считаться дешевле, чем когда это значение около нуля, как результат уменьшения случайного доступа к диску. (Этот столбец содержит NULL, если для типа данных столбца не определён оператор <.)
most_common_elems	anyarray		Список элементов, отличных от NULL, наиболее часто присутствующих в значениях столбца. (NULL для скалярных типов.)
most_common_elem_freqs	real[]		Список частот самых частых элементов, то есть доля строк, содержащих минимум один экземпляр данного значения. За частотами по элементам следуют два или три дополнительных значения: минимум и максимум предшествующих частот по элементам и дополнительно частота элементов NULL. (

Имя	Тип	Ссылки	Описание
			Принимает значение NULL, когда most_common_elems — NULL.)
elem_count_histogram	real[]		Гистограмма количеств различных и отличных от NULL элементов в значениях этого столбца, за которой следует среднее количество элементов, отличных от NULL. (Принимает значение NULL для скалярных типов.)

Максимальным числом записей в полях-массивах можно управлять на уровне столбцов, используя команду ALTER TABLE SET STATISTICS, или глобально, задав параметр времени выполнения default_statistics_target.

49.88. pg_tables

Представление pg_tables даёт доступ к полезной информации обо всех таблицах в базе данных.

Таблица 49.89. Столбцы pg_tables

Имя	Тип	Ссылки	Описание
schemaname	name	pg_namespace .nspname	Имя схемы, содержащей таблицу
tablename	name	pg_class .relname	Имя таблицы
tableowner	name	pg_authid .rolname	Имя владельца таблицы
tablespace	name	pg_tablespace .spcname	Имя табличного пространства, содержащего таблицу (NULL, если это пространство по умолчанию)
hasindexes	boolean	pg_class .relhasindex	True, если эта таблица имеет (или недавно имела) индексы
hasrules	boolean	pg_class .relhasrules	True, если для таблицы определены (или были определены) правила
hastriggers	boolean	pg_class .relhastriggers	True, если для таблицы определены (или были определены) триггеры
rowsecurity	boolean	pg_class .relrowsecurity	True, если для таблицы включена защита строк

49.89. pg_timezone_abbrevs

Представление pg_timezone_abbrevs содержит список аббревиатур часовых поясов, которые в настоящее время распознаются процедурами ввода даты/времени. Содержимое этого представления меняется при изменении параметра времени выполнения timezone_abbreviations.

Таблица 49.90. Столбцы pg_timezone_abbrevs

Имя	Тип	Описание
abbrev	text	Сокращение часового пояса
utc_offset	interval	Смещение от UTC (положительное — к востоку от Гринвича)
is_dst	boolean	True, если эта аббревиатура задаёт летнее время

Большинство аббревиатур часовых поясов представляют фиксированные смещения от UTC, но в некоторые поясах смещение менялось в ходе истории (за подробностями обратитесь к Разделу B.4). В таких случаях данное представление отображает текущее значение.

49.90. pg_timezone_names

В представлении pg_timezone_names содержится список имён часовых поясов, распознаваемых командой SET TIMEZONE, вместе с соответствующими аббревиатурами, смещением UTC и статусом летнего времени. (Говоря технически строго, в Postgres Pro используется не UTC, так как не учитываются секунды координации.) В отличие от аббревиатур, представленных в pg_timezone_abbrevs, многие из этих имён подразумевают некоторый набор правил перехода на летнее время. Поэтому сопутствующая информация меняется при пересечении границ перехода на летнее время. Отображаемая информация вычисляется, исходя из текущего значения CURRENT_TIMESTAMP.

Таблица 49.91. Столбцы pg_timezone_names

Имя	Тип	Описание
name	text	Название часового пояса
abbrev	text	Сокращение часового пояса
utc_offset	interval	Смещение от UTC (положительное — к востоку от Гринвича)
is_dst	boolean	True, если текущие данные отражают летнее время

49.91. pg_user

Представление pg_user открывает доступ к информации о пользователях базы данных. Это просто доступное для всех отображение представления pg_shadow, в котором очищено поле пароля.

Таблица 49.92. Столбцы pg_user

Имя	Тип	Описание
usename	name	Имя пользователя
usesysid	oid	ID этого пользователя
usecreatedb	bool	Пользователь может создавать базы данных
usesuper	bool	Пользователь является суперпользователем
userepl	bool	Пользователь может инициировать потоковую репликацию, включать и отключать режим резервного копирования.

Имя	Тип	Описание
usebypassrls	bool	Пользователь не подчиняется никаким политикам защиты на уровне строк; за подробностями обратитесь к Разделу 5.7.
passwd	text	Не пароль (всегда выводится как ********)
valuntil	abstime	Срок действия пароля (используется только при аутентификации по паролю)
useconfig	text[]	Сеансовые значения по умолчанию для переменных конфигурации времени выполнения

49.92. pg_user_mappings

Представление pg_user_mappings открывает доступ к информации о сопоставлениях пользователей. По сути это просто доступное для чтения всеми отображение pg_user_mapping, в котором поле параметров показывается, только если у пользователя есть права читать его.

Таблица 49.93. Столбцы pg_user_mappings

Имя	Тип	Ссылки	Описание
umid	oid	pg_user_mapping .oid	OID сопоставления пользователей
srvid	oid	pg_foreign_server .oid	OID стороннего сервера, содержащего это сопоставление
srvname	name	pg_foreign_server .srvname	Имя стороннего сервера
umuser	oid	pg_authid .oid	OID сопоставляемой локальной роли, либо 0, если сопоставление задаётся для всех
usename	name		Имя локального пользователя, для которого задано сопоставление
umoptions	text[]		Специальные параметры сопоставления пользователей, в виде строк «ключ=значение»

Для защиты информации о паролях, хранящейся в свойствах сопоставления пользователей, столбец umoptions при чтении будет содержать не NULL, если только выполняется одно из этих условий:

- текущий пользователь является объектом сопоставления и владельцем сервера или имеет право USAGE для него
- текущий пользователь является владельцем сервера и прочитывается сопоставление для PUBLIC

- текущий пользователь является суперпользователем

49.93. pg_views

Представление pg_views даёт доступ к полезной информации обо всех представлениях в базе данных.

Таблица 49.94. Столбцы pg_views

Имя	Тип	Ссылки	Описание
schemaname	name	pg_namespace .nspname	Имя схемы, содержащей представление
viewname	name	pg_class .relname	Имя представления
viewowner	name	pg_authid .rolname	Имя владельца представления
definition	text		Определение представления (реконструированный запрос SELECT)

Глава 50. Клиент-серверный протокол

Клиенты и серверы Postgres Pro взаимодействуют друг с другом, используя специальный протокол, основанный на сообщениях. Этот протокол поддерживается для соединений по TCP/IP и через Unix-сокеты. Для серверов, поддерживающих этот протокол, в IANA зарезервирован номер TCP-порта 5432, но на практике можно задействовать любой порт, не требующий особых привилегий.

В этой документации описана версия 3.0 этого протокола, реализованная в PostgreSQL версии 7.4 и новее. За описанием предыдущих версий протокола обратитесь к документации более ранних выпусков PostgreSQL. Один сервер способен поддерживать несколько версий протокола. Какую версию протокола пытается использовать клиент, сервер узнаёт из стартового сообщения при установлении соединения. Если старшая версия, запрашиваемая клиентом, не поддерживается сервером, соединение будет разорвано (например, это будет иметь место, если клиент запросит протокол версии 4.0, несуществующий на момент написания этого текста). Если младшая версия, запрашиваемая клиентом, не поддерживается сервером (например, клиент запросил версию 3.1, а сервер поддерживает только 3.0), сервер может либо разорвать соединение, либо ответить сообщением NegotiateProtocolVersion с указанием наибольшей младшей версии, которую он поддерживает. Затем клиент может решить либо продолжить установление соединения с указанной версией протокола, либо разорвать соединение.

Чтобы эффективно обслуживать множество клиентов, сервер запускает отдельный «обслуживающий» процесс для каждого клиента. В текущей реализации новый дочерний процесс запускается немедленно после обнаружения входящего подключения. Однако это происходит прозрачно для протокола. С точки зрения протокола, термины «обслуживающий процесс», «процесс заднего плана» и «сервер» взаимозаменяемы, как и «приложение переднего плана» и «клиент».

50.1. Обзор

В протоколе выделены отдельные фазы для запуска и обычных операций. На стадии запуска клиент устанавливает подключение к серверу и должен удовлетворить сервер, подтвердив свою подлинность. (Для этого может потребоваться одно или несколько сообщений, в зависимости от применяемого метода проверки подлинности.) Если всё проходит успешно, сервер сообщает клиенту о текущем состоянии, а затем переходит к обычной работе. Не считая начального стартового сообщения, в этой фазе протокола ведущую роль играет сервер.

В ходе обычной работы клиент передаёт запросы и другие команды серверу, а сервер возвращает результаты запросов и другие ответы. В некоторых случаях (например, с `NOTIFY`) сервер передаёт клиенту сообщения по своей инициативе, но по большей части эта фаза сеанса управляется запросами клиента.

Завершение сеанса обычно происходит по желанию клиента, но в некоторых случаях и сервер может принудительно завершить сеанс. В любом случае, когда сервер закрывает соединение, он предварительно откатывает любую открытую (незавершённую) транзакцию.

В процессе обычной работы команды SQL могут выполняться по одному из двух внутренних протоколов. По протоколу «простых запросов» клиент посылает просто текстовую строку запроса, которую сервер сразу же разбирает и выполняет. С протоколом «расширенных запросов» обработка запросов разделяется на несколько этапов: разбор, привязывание значений параметров и исполнение. Это даёт дополнительную гибкость и может увеличить быстродействие ценой большей сложности.

В обычном режиме также поддерживаются дополнительные внутренние протоколы для специальных операций, например `COPY`.

50.1.1. Обзор обмена сообщениями

Всё взаимодействие представляет собой поток сообщений. Первый байт сообщения определяет тип сообщения, а следующие четыре байта задают длину остального сообщения (эта длина

включает размер самого поля длины, но не байт с типом сообщения). Остальное содержимое сообщения определяется его типом. По историческим причинам в самом первом сообщении, передаваемом клиентом, (стартовом сообщении) первый байт с типом сообщения отсутствует.

Чтобы не потерять синхронизацию в потоке сообщений, и серверы, и клиенты обычно считывают всё сообщение в буфер (его размер определяется счётчиком байт), прежде чем обрабатывать его содержимое. Это позволяет без труда продолжить работу, если возникает ошибка при разборе сообщения. В исключительных случаях (например, при нехватке памяти для помещения сообщения в буфер), счётчик байт помогает получателю определить, сколько поступающих байт нужно пропустить, прежде чем продолжать получать сообщения.

С другой стороны, и клиенты, и серверы, ни при каких условиях не должны передавать неполные сообщения. Чтобы этого не допустить, обычно всё сообщение сначала размещается в буфере, и только потом передаётся. Если в процессе отправки или получения сообщения происходит сбой передачи, единственным разумным вариантом продолжения будет прерывание соединения, так как вероятность восстановления синхронизации по границам сообщений в этой ситуации минимальна.

50.1.2. Обзор расширенных запросов

В протоколе расширенных запросов исполнение команд SQL разделяется на несколько этапов. Состояние между этапами представляется объектами двух типов: *подготовленные операторы* и *порталы*. Подготовленный оператор представляет собой результат разбора и семантического анализа текстовой строки запроса. Подготовленный оператор сам по себе не готов для исполнения, так как он может не иметь конкретных значений для *параметров*. Портал представляет собой готовый к исполнению или уже частично выполненный оператор, в котором заданы все недостающие значения параметров. (Для операторов `SELECT` портал равнозначен открытому курсору, но мы выбрали другой термин, так как курсоры неприменимы к операторам, отличным от `SELECT`.)

Общий цикл выполнения состоит из этапа *разбора*, на котором из текстовой строки запроса создаётся подготовленный оператор; этапа *привязки*, на котором из подготовленного оператора и значений для необходимых параметров создаётся портал; и этапа *выполнения*, на котором исполняется запрос портала. В случае запроса, возвращающего строки (`SELECT`, `SHOW` и т. д.), можно указать, чтобы за один шаг выполнения возвращалось только ограниченное число строк, так что для завершения операции понадобятся несколько шагов выполнения.

Сервер может контролировать одновременно несколько подготовленных операторов и порталов (но учтите, что они существуют только в рамках сеанса и никогда не разделяются между сеансами). Обращаться к подготовленным операторам и порталам можно по именам, которые назначаются им при создании. Кроме того, существуют и «безымянные» подготовленные операторы и порталы. Хотя они практически не отличаются от именованных объектов, операции с ними оптимизированы для разового выполнения запроса с последующим освобождением объекта, тогда как операции с именованными объектами оптимизируются в расчёте на многоразовое использование.

50.1.3. Форматы и коды форматов

Данные определённого типа могут передаваться в одном из нескольких различных *форматов*. С версии 7.4 PostgreSQL поддерживаются только текстовый («text») и двоичный («binary») форматы, но в протоколе предусмотрены возможности для расширения в будущем. Ожидаемый формат для любого значения задаётся *кодом формата*. Клиенты могут указывать код формата для каждого передаваемого значения параметра и для каждого столбца результата запроса. Текстовый формат имеет код ноль, двоичный — код один, а другие коды оставлены для определения в будущем.

Текстовым представлением значений будут строки, которые выдаются и принимаются функциями ввода/вывода определённого типа данных. В передаваемом представлении завершающий нулевой символ отсутствует, клиент должен добавить его сам, если хочет обрабатывать такое представление в виде строки C. (Собственно, данные в текстовом формате не могут содержать нулевые символы.)

В двоичном представлении целых чисел применяется сетевой порядок байт (наиболее значащий байт первый). Какое именно двоичное представление имеют другие типы данных, можно узнать в документации или исходном коде. Но учтите, что двоичное представление сложных типов данных может меняться от версии к версии сервера; с точки зрения портируемости обычно лучше текстовый формат.

50.2. Поток сообщений

В этом разделе описывается поток сообщений и семантика каждого типа сообщений. (Подробнее точное представление каждого сообщения описывается в Разделе 50.7.) В зависимости от состояния соединения выделяются несколько различных подразделов протокола: запуск, запрос, вызов функции, копирование (`COPY`) и завершение. Есть также специальные средства для асинхронных операций (в частности, для уведомлений и отмены команд), которые могут выполняться в любой момент после этапа запуска.

50.2.1. Запуск

Чтобы начать сеанс, клиент открывает подключение к серверу и передаёт стартовое сообщение. В этом сообщении содержатся имена пользователя и базы данных, к которой пользователь хочет подключиться; в нём также определяется, какая именно версия протокола будет использоваться. (Стартовое сообщение также может содержать дополнительные значения для параметров времени выполнения.) Проанализировав эту информацию и содержимое своих файлов конфигурации (в частности, `pg_hba.conf`), сервер определяет, можно ли предварительно разрешить это подключение, и какая дополнительная проверка подлинности требуется.

Затем сервер отправляет соответствующее сообщение с запросом аутентификации, на которое клиент должен ответить сообщением, подтверждающим его подлинность (например, по паролю). Для всех методов аутентификации, за исключением GSSAPI, SSPI и SASL, может быть максимум один запрос и один ответ. Для некоторых методов ответ клиента вообще не требуется, так что запрос аутентификации также не передаётся. Методы GSSAPI, SSPI и SASL для прохождения проверки подлинности могут потребовать выполнить серию обменов пакетами.

Цикл аутентификации заканчивает сервер, либо запрещая соединение (ErrorResponse), либо принимая его (отправляя AuthenticationOk).

Сервер может передавать в этой фазе следующие сообщения:

ErrorResponse (Ошибочный ответ)

Попытка соединения была отвергнута. Сразу после этого сервер закрывает соединение.

AuthenticationOk (Аутентификация пройдена)

Обмен сообщениями для проверки подлинности завершён успешно.

AuthenticationKerberosV5 (Аутентификация Kerberos V5)

Клиент должен теперь принять участие в диалоге аутентификации по протоколу Kerberos V5 (здесь его детали не описывается, так как они относятся к спецификации Kerberos) с сервером. Если этот диалог завершается успешно, сервер отвечает AuthenticationOk, иначе — ErrorResponse. Этот вариант аутентификации больше не поддерживается.

AuthenticationCleartextPassword (Аутентификация с открытым паролем)

Клиент должен передать в ответ сообщение PasswordMessage, содержащее пароль в открытом виде. Если пароль правильный, сервер отвечает ему AuthenticationOk, иначе — ErrorResponse.

AuthenticationMD5Password (Аутентификация с паролем MD5)

Клиент должен передать в ответ сообщение PasswordMessage с результатом преобразования пароля (и имени пользователя) в хеш MD5 с последующим хешированием с четырёхбайтовым случайным значением соли, переданным в сообщении AuthenticationMD5Password. Если пароль правильный, сервер отвечает AuthenticationOk, иначе — ErrorResponse.

Содержимое сообщения PasswordMessage можно вычислить в SQL как `concat('md5', md5(concat(md5(concat(password, username)), random-salt)))`. (Учтите, что функция `md5()` возвращает результат в виде шестнадцатеричной строки.)

AuthenticationSCMCredential (Аутентификация по учётным данным SCM)

Этот ответ возможен только для локальных подключений через Unix-сокеты на платформах, поддерживающих сообщения с учётными данными SCM. Клиент должен выдать сообщение с учётными данными SCM и дополнительно отправить один байт данных. (Содержимое этого байта не представляет интереса; его нужно передавать, только чтобы сервер дожидался сообщения с учётными данными.) Если сервер принимает учётные данные, он отвечает AuthenticationOk, иначе — ErrorResponse. (Этот тип сообщений выдают только серверы версии до 9.1. В конце концов он может быть исключён из спецификации протокола.)

AuthenticationGSS (Аутентификация GSS)

Клиент должен начать согласование GSSAPI. В ответ на это сообщение клиент отправляет GSSResponse с первой частью потока данных GSSAPI. Если потребуются дополнительные сообщения, сервер передаст в ответ AuthenticationGSSContinue.

AuthenticationSSPI (Аутентификация SSPI)

Клиент должен начать согласование SSPI. В ответ на это сообщение клиент отправляет GSSResponse с первой частью потока данных SSPI. Если потребуются дополнительные сообщения, сервер передаст в ответ AuthenticationGSSContinue.

AuthenticationGSSContinue (Продолжение аутентификации GSS)

Это сообщение содержит данные ответа на предыдущий шаг согласования GSSAPI или SSPI (AuthenticationGSS, AuthenticationSSPI или предыдущего AuthenticationGSSContinue). Если в структуре GSSAPI или SSPI в этом сообщении указывается, что для завершения аутентификации требуются дополнительные данные, клиент должен передать их в очередном сообщении GSSResponse. Если этим сообщением завершается проверка подлинности GSSAPI или SSPI, сервер затем передаёт AuthenticationOk, сообщая об успешной проверке подлинности, либо ErrorResponse, сообщая об ошибке.

AuthenticationSASL (Аутентификация SASL)

Клиент должен начать согласование SASL, используя один из механизмов SASL, перечисленных в сообщении. В ответ на это сообщение клиент отправляет SASLInitialResponse с именем выбранного механизма и первой частью потока данных SASL. Если потребуются дополнительные сообщения, сервер передаст в ответ AuthenticationSASLContinue. За подробностями обратитесь к Разделу 50.3.

AuthenticationSASLContinue (Продолжение аутентификации SASL)

Это сообщение содержит данные вызова с предыдущего шага согласования SASL (AuthenticationSASL или предыдущего AuthenticationSASLContinue). Клиент должен передать в ответ сообщение SASLResponse.

AuthenticationSASLFinal (Окончание аутентификации SASL)

Аутентификация SASL завершена с дополнительными данными для клиента, специфичными для механизма. Затем сервер передаст сообщение AuthenticationOk, говорящее об успешной аутентификации, или ErrorResponse, говорящее об ошибке. Данное сообщение передаётся, только если механизм SASL должен в завершение передать с сервера клиенту дополнительные специфичные данные.

NegotiateProtocolVersion

Сервер не поддерживает младшую версию протокола, запрошенную клиентом, но поддерживает более раннюю версию протокола; в этом сообщении указывается наибольшая поддерживаемая младшая версия. Это сообщение будет также передаваться, если клиент

запросил в стартовом пакете неподдерживаемые параметры протокола (то есть, начинающиеся с `_pq_`.). За этим сообщением должен последовать или ответ ErrorResponse, или сообщение, говорящее об успехе или неудаче проверки подлинности.

Если клиент не поддерживает метод проверки подлинности, запрошенный сервером, он должен немедленно закрыть соединение.

Получив сообщение AuthenticationOk, клиент должен ждать дальнейших сообщений от сервера. В этой фазе запускается обслуживающий процесс, а клиент представляет собой просто заинтересованного наблюдателя. Попытка запуска может быть неудачной (и клиент получит ErrorResponse) либо сервер может отказать в поддержке запрошенной младшей версии протокола (NegotiateProtocolVersion), но в обычной ситуации обслуживающий процесс передаёт несколько сообщений ParameterStatus, BackendKeyData и, наконец, ReadyForQuery.

В ходе этой фазы обслуживающий процесс попытается применить все параметры времени выполнения, полученные в стартовом сообщении. Если это удастся, эти значения становятся сеансовыми значениями по умолчанию. При ошибке он передаёт ErrorResponse и завершается.

Обслуживающий процесс может передавать в этой фазе следующие сообщения:

BackendKeyData (Данные ключа сервера)

В этом сообщении передаётся секретный ключ, который клиент должен сохранить, чтобы впоследствии иметь возможность выполнять запросы. Клиент не должен отвечать на это сообщение, он должен дожидаться сообщения ReadyForQuery.

ParameterStatus (Состояние параметров)

Это сообщение говорит клиенту о текущих (начальных) значениях параметров обслуживающего процесса, например, client_encoding или DateStyle. Клиент может проигнорировать это сообщение или сохранить значения для дальнейшего использования; за дополнительными подробностями обратитесь к Подразделу 50.2.7. Клиент не должен отвечать на это сообщение, он должен дожидаться сообщения ReadyForQuery.

ReadyForQuery (Готов к запросам)

Запуск завершён. Теперь клиент может выполнять команды.

ErrorResponse (Ошибочный ответ)

Запуск не удался. Соединение закрывается после передачи этого сообщения.

NoticeResponse (Ответ с замечанием)

Выдаётся предупреждающее сообщение. Клиент должен вывести это сообщение, но продолжать ожидать сообщения ReadyForQuery или ErrorResponse.

Сообщение ReadyForQuery в данной фазе ничем не отличается от сообщений, который передаёт сервер после каждого цикла команд. В зависимости от условий реализации клиента, можно воспринимать сообщение ReadyForQuery как начинающее цикл команд, либо как завершающее фазу запуска и каждый последующий цикл команд.

50.2.2. Простой запрос

Цикл простого запроса начинает клиент, передавая серверу сообщение Query. Это сообщение включает команду (или команды) SQL, выраженную в виде текстовой строки. В ответ сервер передаёт одно или несколько сообщений, в зависимости от строки запроса, и завершает цикл сообщением ReadyForQuery. ReadyForQuery говорит клиенту, что он может безопасно передавать новую команду. (На самом деле клиент может передать следующую команду, не дожидаясь ReadyForQuery, но тогда он сам должен разобраться в ситуации, когда первая команда завершается ошибкой, а последующая выполняется успешно.)

Сервер может передавать в этой фазе следующие ответные сообщения:

CommandComplete (Команда завершена)

Команда SQL выполнена нормально.

CopyInResponse (Ответ входящего копирования)

Сервер готов копировать данные, получаемые от клиента, в таблицу; см. Подраздел 50.2.6.

CopyOutResponse (Ответ исходящего копирования)

Сервер готов копировать данные из таблицы клиенту; см. Подраздел 50.2.6.

RowDescription (Описание строк)

Показывает, что в ответ на запрос `SELECT`, `FETCH` и т. п. будут возвращены строки. В содержимом этого сообщения описывается структура столбцов этих строк. За ним для каждой строки, возвращаемой клиенту, следует сообщение DataRow.

DataRow (Строка данных)

Одна строка из набора, возвращаемого запросом `SELECT`, `FETCH` и т. п.

EmptyQueryResponse (Ответ на пустой запрос)

Была принята пустая строка запроса.

ErrorResponse (Ошибочный ответ)

Произошла ошибка.

ReadyForQuery (Готов к запросам)

Обработка строки запроса завершена. Чтобы отметить это, отправляется отдельное сообщение, так как строка запроса может содержать несколько команд SQL. (Сообщение CommandComplete говорит о завершении обработки одной команды SQL, а не всей строки.) ReadyForQuery передаётся всегда, и при успешном завершении обработки, и при ошибке.

NoticeResponse (Ответ с замечанием)

Выдаётся предупреждение, связанное с запросом. Эти замечания дополняют другие ответы, то есть сервер, выдавая их, продолжает обрабатывать команду.

Ответ на запрос `SELECT` (или другие запросы, возвращающие наборы строк, такие как `EXPLAIN` и `SHOW`) обычно состоит из RowDescription, нуля или нескольких сообщений DataRow, и завершающего CommandComplete. Для команды `COPY` с вводом или выводом данных через клиента, применяется специальный протокол, описанный в Подразделе 50.2.6. Со всеми другими типами запросами обычно выдаётся только сообщение CommandComplete.

Так как строка запроса может содержать несколько запросов (разделённых точкой с запятой), до завершения обработки всей строки сервер может передать несколько серий таких ответов. Когда сервер завершает обработку всей строки и готов принять следующую строку запроса, он выдаёт сообщение ReadyForQuery.

Если получена полностью пустая строка запроса (не содержащая ничего, кроме пробельных символов), ответом будет EmptyQueryResponse с последующим ReadyForQuery.

В случае ошибки выдаётся ErrorResponse с последующим ReadyForQuery. Сообщение ErrorResponse прерывает дальнейшую обработку строки запроса (даже если в ней остались другие запросы). Заметьте, что оно может быть выдано и в середине последовательности сообщений, выдаваемых в ответ на отдельный запрос.

В режиме простых запросов получаемые значения всегда передаются в текстовом формате, за исключением результатов команды `FETCH` для курсора, объявленного с атрибутом `BINARY`. С такой командой значения передаются в двоичном формате. Какой именно формат используется, определяют коды формата, передаваемые в сообщении RowDescription.

Клиент должен быть готов принять сообщения ErrorResponse и NoticeResponse, ожидая любой другой тип сообщений. Также обратитесь к Подразделу 50.2.7 за информацией о сообщениях, которые сервер может выдавать в ответ на внешние события.

Код клиента рекомендуется реализовывать в виде конечного автомата, который в любой момент будет принимать сообщения всех типов, имеющих смысл на данном этапе, но не программировать жёстко обработку точной последовательности сообщений.

50.2.3. Расширенный запрос

Протокол расширенных запросов разбивает вышеописанный протокол простых запросов на несколько шагов. Результаты подготовительных шагов можно неоднократно использовать повторно для улучшения эффективности. Кроме того, он открывает дополнительные возможности, в частности, возможность передавать значения данных в отдельных параметрах вместо того, чтобы внедрять их непосредственно в строку запроса.

В расширенном протоколе клиент сначала передаёт сообщение Parse с текстовой строкой запроса и, возможно, некоторыми сведениями о типах параметров и именем целевого объекта подготовленного оператора (если имя пустое, создаётся безымянный подготовленный оператор). Ответом на это сообщение будет ParseComplete или ErrorResponse. Типы параметров указываются по OID; при отсутствии явного указания анализатор запроса пытается определить типы данных так же, как он делал бы для нетипизированных строковых констант.

Примечание

Тип данных параметра можно оставить неопределённым, задав для него значение ноль, либо сделав массив с OID типов параметров короче, чем набор символов параметров (`$n`), используемых в строке запроса. Другой особый случай — передача типа параметра как `void` (то есть передача OID псевдотипа `void`). Это предусмотрено для того, чтобы символы параметров можно было использовать для параметров функций, на самом деле представляющих собой параметры OUT. Обычно параметр `void` нельзя использовать ни в каком контексте, но если такой параметр фигурирует в списке параметров функции, он фактически игнорируется. Например, вызову функции `foo($1,$2,$3,$4)` может соответствовать функция с аргументами IN и двумя OUT, если аргументы `$3` и `$4` объявлены как имеющие тип `void`.

Примечание

Строка запроса, содержащаяся в сообщении Parse, не может содержать больше одного оператора SQL; иначе выдаётся синтаксическая ошибка. Это ограничение отсутствует в протоколе простых запросов, но присутствует в расширенном протоколе, так как добавление поддержки подготовленных операторов или порталов, содержащих несколько команд, неоправданно усложнило бы протокол.

В случае успешного создания именованный подготовленный оператор продолжает существовать до завершения текущего сеанса, если только он не будет уничтожен явно. Безымянный подготовленный оператор сохраняется только до следующей команды Parse, в которой целевым является безымянный оператор. (Заметьте, что сообщение простого запроса также уничтожает безымянный оператор.) Именованные операторы должны явно закрываться, прежде чем их можно будет переопределить другим сообщением Parse, но для безымянных операторов это не требуется. Именованные подготовленные операторы также можно создавать и вызывать на уровне команд SQL, используя команды `PREPARE` и `EXECUTE`.

Когда подготовленный оператор существует, его можно подготовить к выполнению сообщением Bind. В сообщении Bind задаётся имя исходного подготовленного оператора (пустая строка подразумевает безымянный подготовленный оператор), имя целевого портала (пустая строка

подразумевает безымянный портал) и значения для любых шаблонов параметров, представленных в подготовленном операторе. Набор передаваемых значений должен соответствовать набору параметров, требующихся для подготовленного оператора. (Если вы объявили параметры `void` в сообщении Parse, передайте для них значения NULL в сообщении Bind.) Bind также принимает указание формата для данных, возвращаемых в результате запроса; формат можно указать для всех данных, либо для отдельных столбцов. Ответом на это сообщение будет BindComplete или ErrorResponse.

> **Примечание**
>
> Выбор между текстовым и двоичным форматом вывода определяется кодами формата, передаваемыми в Bind, вне зависимости от команды SQL. При использовании протокола расширенных запросов атрибут `BINARY` в объявлении курсоров не имеет значения.

Планирование запроса обычно имеет место при обработке сообщения Bind. Если подготовленный оператор не имеет параметров, либо он выполняется многократно, сервер может сохранить созданный план и использовать его повторно при последующих сообщениях Bind для того же подготовленного оператора. Однако он будет делать это, только если решит, что можно получить универсальный план, который не будет значительно неэффективнее планов, зависящих от конкретных значений параметров. С точки зрения протокола это происходит незаметно.

В случае успешного создания объект именованного портала продолжает существование до конца текущей транзакции, если только он не будет уничтожен явно. Безымянный портал уничтожается в конце транзакции или при выполнении следующей команды Bind, в которой в качестве целевого выбирается безымянный портал. (Заметьте, что сообщение простого запроса также уничтожает безымянный портал.) Именованные порталы должны явно закрываться, прежде чем их можно будет явно переопределить другим сообщением Bind, но это не требуется для безымянных порталов. Именованные порталы также можно создавать и вызывать на уровне команд SQL, используя команды `DECLARE CURSOR` и `FETCH`.

Когда портал существует, его можно запустить на выполнение сообщением Execute. В сообщении Execute указывается имя портала (пустая строка подразумевает безымянный портал) и максимальное число результирующих строк (ноль означает «выбрать все строки»). Число результирующих строк имеет значение только для порталов, которые содержат команды, возвращающие наборы строк; в других случаях команда всегда выполняется до завершения и число строк игнорируется. В ответ на Execute могут быть получены те же сообщения, что описаны выше для запросов, выполняемых через протокол простых запросов, за исключением того, что после Execute не выдаются сообщения ReadyForQuery и RowDescription.

Если операция Execute оканчивается до завершения выполнения портала (из-за достижения ненулевого ограничения на число строк), сервер отправляет сообщение PortalSuspended; появление этого сообщения говорит клиенту о том, что для завершения операции с данным порталом нужно выдать ещё одно сообщение Execute. Сообщение CommandComplete, говорящее о завершении исходной команды SQL, не передаётся до завершения выполнения портала. Таким образом, фаза Execute всегда заканчивается при появлении одного из сообщений: CommandComplete, EmptyQueryResponse (если портал был создан из пустой строки запроса), ErrorResponse или PortalSuspended.

В конце каждой серии сообщений расширенных запросов клиент должен выдать сообщение Sync. Получив это сообщение без параметров, сервер закрывает текущую транзакцию, если команды выполняются не внутри блока транзакции `BEGIN`/`COMMIT` (под «закрытием» понимается фиксация при отсутствии ошибок или откат в противном случае). Затем он выдаёт ответ ReadyForQuery. Целью сообщения Sync является обозначение точки синхронизации для восстановления в случае ошибок. Если при обработке сообщений расширенных запросов происходит ошибка, сервер выдаёт ErrorResponse, затем считывает и пропускает сообщения до Sync, после чего выдаёт ReadyForQuery и возвращается к обычной обработке сообщений. (Но заметьте, что он не будет пропускать следующие сообщения, если ошибка происходит *в процессе* обработки Sync — это гарантирует, что для каждого Sync будет передаваться в точности одно сообщение ReadyForQuery.)

> **Примечание**
>
> Сообщение Sync не приводит к закрытию блока транзакции, открытого командой `BEGIN`. Выявить эту ситуацию можно, используя информацию о состоянии транзакции, содержащуюся в сообщении ReadyForQuery.

В дополнение к этим фундаментальным и обязательным операциям, протокол расширенных запросов позволяет выполнить и несколько дополнительных операций.

В сообщении Describe (в вариации для портала) задаётся имя существующего портала (пустая строка обозначает безымянный портал). В ответ передаётся сообщение RowDescription, описывающее строки, которые будут возвращены при выполнении портала; либо сообщение NoData, если портал не содержит запроса, возвращающего строки; либо ErrorResponse, если такого портала нет.

В сообщении Describe (в вариации для оператора) задаётся имя существующего подготовленного оператора (пустая строка обозначает безымянный подготовленный оператор). В ответ передаётся сообщение ParameterDescription, описывающее параметры, требующиеся для оператора, за которым следует сообщение RowDescription, описывающее строки, которые будут возвращены, когда оператор будет собственно выполнен (или сообщение NoData, если оператор не возвратит строки). ErrorResponse выдаётся, если такой подготовленный оператор отсутствует. Заметьте, что так как команда Bind не выполнялась, сервер ещё не знает, в каком формате будут возвращаться столбцы; в этом случае поля кодов формата в сообщении RowDescription будут содержать нули.

> **Подсказка**
>
> В большинстве случаев клиент должен выдать ту или иную вариацию Describe, прежде чем выдавать Execute, чтобы понять, как интерпретировать результаты, которые он получит.

Сообщение Close закрывает существующий подготовленный оператор или портал и освобождает связанные ресурсы. При попытке выполнить Close для имени несуществующего портала или оператора ошибки не будет. Ответ на это сообщение обычно CloseComplete, но может быть и ErrorResponse, если при освобождении ресурсов возникают проблемы. Заметьте, что при закрытии подготовленного оператора неявно закрываются все открытые порталы, которые были получены из этого оператора.

Сообщение Flush не приводит к генерации каких-либо данных, а указывает серверу передать все данные, находящиеся в очереди в его буферах вывода. Сообщение Flush клиент должен отправлять после любой команды расширенных запросов, кроме Sync, если он желает проанализировать результаты этой команды, прежде чем выдавать следующие команды. Без Flush сообщения, возвращаемые сервером, будут объединяться вместе в минимальное количество пакетов с целью уменьшения сетевого трафика.

> **Примечание**
>
> Простое сообщение Query примерно равнозначно последовательности сообщений Parse, Bind, Describe (для портала), Execute, Close, Sync, с использованием объектов подготовленного оператора и портала без имён и без параметров. Одно отличие состоит в том, что такое сообщение может содержать в строке запроса несколько операторов SQL, для каждого из которых по очереди автоматически выполняется последовательность Bind/Describe/Execute. Другое отличие состоит в том, что в ответ на него не приходят сообщения ParseComplete, BindComplete, CloseComplete или NoData.

50.2.4. Конвейеризация

Протокол расширенных запросов позволяет использовать *конвейеризацию*, то есть отправлять серию запросов, не ожидая завершения каждого предыдущего. При этом уменьшается количество сетевых операций, необходимых для выполнения данной последовательности запросов. Однако пользователь должен тщательно продумывать поведение в случае сбоя одного из шагов, поскольку последующие запросы уже будут отправлены на сервер.

Например, можно объединить всю последовательность запросов в одну транзакцию, заключив её в `BEGIN` ... `COMMIT`. Однако если какие-то команды должны выполняться независимо от остальных, этот способ не подходит.

Протокол расширенных запросов позволяет реализовать поведение в случае ошибки по-другому, исключив передачу сообщений Sync между зависимыми шагами. Поскольку после ошибки сервер пропускает командные сообщения до тех пор, пока не получит Sync, более поздние команды в конвейере автоматически пропускаются при сбое более ранней, при этом клиенту не нужно явно управлять этим с помощью `BEGIN` и `COMMIT`. Сегменты конвейера, фиксируемые независимо, можно разделять сообщениями Sync.

Если клиент не передаёт команду `BEGIN` явно, то каждое сообщение Sync обычно вызывает неявное выполнение `COMMIT` в случае успешного завершения предыдущих шагов или `ROLLBACK` в случае сбоя. Однако есть несколько команд DDL (например, `CREATE DATABASE`), которые нельзя выполнить внутри блока транзакции. Если такая команда выполняется в конвейере, в случае успешного выполнения она вызовет немедленную фиксацию для сохранения согласованности базы данных. Если сразу за ней отправить Sync, никаких дополнительных действий это сообщение не вызовет, кроме ответа ReadyForQuery.

Применяя этот метод, факт завершения конвейера нужно определять, подсчитывая сообщения ReadyForQuery и ожидая достижения количества отправленных сообщений Sync. Вариант с подсчётом сообщений о завершении команд ненадёжен, так как некоторые команды могут не выполняться, так что такие сообщения выдаваться не будут.

50.2.5. Вызов функций

Подраздел протокола «Вызов функций» позволяет клиенту запросить непосредственный вызов любой функции, существующей в системном каталоге `pg_proc`. При этом клиент должен иметь право на выполнение этой функции.

> **Примечание**
>
> Этот подраздел протокола считается устаревшим и в новом коде использовать его не следует. Примерно тот же результат можно получить, подготовив оператор с командой `SELECT function($1, ...)`. При таком подходе цикл вызова функции заменяется последовательностью Bind/Execute.

Цикл вызова функции начинает клиент, передавая серверу сообщение FunctionCall. Сервер возвращает одно или несколько сообщений ответа, в зависимости от результата вызова функции, и завершающее сообщение ReadyForQuery. ReadyForQuery говорит клиенту, что он может свободно передавать новый запрос или вызов функции.

Сервер может передавать в этой фазе следующие ответные сообщения:

ErrorResponse (Ошибочный ответ)

Произошла ошибка.

FunctionCallResponse (Ответ на вызов функции)

Вызов функции завершён и в этом сообщении передаётся её результат. (Заметьте, что протокол вызова функций позволяет выдать только один скалярный результат, но не кортеж или набор результатов.)

ReadyForQuery (Готов к запросам)

Обработка вызова функции завершена. В ответ всегда передаётся ReadyForQuery, независимо от того, была ли функция выполнена успешно или с ошибкой.

NoticeResponse (Ответ с замечанием)

Выдаётся предупреждение, связанное с вызовом функции. Эти замечания дополняют другие ответы, то есть сервер, выдавая их, продолжает обрабатывать вызов.

50.2.6. Операции COPY

Команда `COPY` позволяет обеспечить скоростную передачу данных на сервер или с сервера. Операции входящего и исходящего копирования переключают соединение в отдельные режимы протокола, которые завершаются только в конце операции.

Режим входящего копирования (передача данных на сервер) включается, когда клиент выполняет SQL-оператор `COPY FROM STDIN`. Переходя в этот режим, сервер передаёт клиенту сообщение CopyInResponse. После этого клиент должен передать ноль или более сообщений CopyData, образующих поток входных данных. (При этом границы сообщений не обязательно должны совпадать с границами строк данных, хотя часто имеет смысл выровнять их.) Клиент может завершить режим входящего копирования, передав либо сообщение CopyDone (говорящее об успешном завершении), либо CopyFail (которое приведёт к завершению SQL-оператора `COPY` с ошибкой). При этом сервер вернётся в обычный режим обработки, в котором он находился до выполнения команды `COPY` (это может быть протокол простых или расширенных запросов). Затем он отправит сообщение CommandComplete (в случае успешного завершения) или ErrorResponse (в противном случае).

В случае возникновения ошибки в режиме входящего копирования (включая получение сообщения CopyFail), сервер выдаёт сообщение ErrorResponse. Если команда `COPY` была получена в сообщении расширенного запроса, сервер не будет обрабатывать последующие сообщения клиента, пока не получит сообщение Sync, после которого он выдаст ReadyForQuery и вернётся в обычный режим работы. Если команда `COPY` была получена в сообщении простого запроса, остальная часть сообщения игнорируется и сразу выдаётся ReadyForQuery. В любом случае все последующие сообщения CopyData, CopyDone или CopyFail, поступающие от клиента, будут просто игнорироваться.

В режиме входящего копирования сервер игнорирует поступающие сообщения Flush и Sync. При поступлении сообщений любого другого типа, не связанного с копированием, возникает ошибка, приводящая к прерыванию режима входящего копирования, как описано выше. (Исключение для сообщений Flush и Sync сделано для удобства клиентских библиотек, которые всегда передают Flush или Sync после сообщения Execute, не проверяя, не запускается ли в нём команда `COPY FROM STDIN`.)

Режим исходящего копирования (передача данных с сервера) включается, когда клиент выполняет SQL-оператор `COPY TO STDOUT`. Переходя в этот режим, сервер передаёт клиенту сообщение CopyOutResponse, за ним ноль или более сообщений CopyData (всегда одно сообщение для каждой строки) и в завершение CopyDone. Затем сервер возвращается в обычный режим обработки, в котором он находился до выполнения команды `COPY`, и передаёт CommandComplete. Клиент не может прервать передачу (кроме как закрыв соединение или выдав запрос Cancel), но он может игнорировать ненужные ему сообщения CopyData и CopyDone.

В случае обнаружения ошибки в режиме исходящего копирования, сервер выдаёт сообщение ErrorResponse и возвращается к обычной обработке. Клиент должен воспринимать поступление ErrorResponse как завершение режима исходящего копирования.

Между сообщениями CopyData могут поступать сообщения NoticeResponse и ParameterStatus; клиенты должны обрабатывать их и быть готовы принимать и другие типы асинхронных сообщений (см. Подраздел 50.2.7). В остальном, сообщения любых типов, кроме CopyData и CopyDone, могут восприниматься как завершающие режим исходящего копирования.

Есть ещё один режим копирования, называемый двусторонним копированием и обеспечивающий высокоскоростную передачу данных на *и* с сервера. Двустороннее копирование запускается, когда клиент в режиме walsender выполняет оператор `START_REPLICATION`. В ответ сервер передаёт клиенту сообщение CopyBothResponse. Затем и сервер, и клиент могут передавать друг другу сообщения CopyData, пока кто-то из них не завершит передачу сообщением CopyDone. Когда сообщение CopyDone передаёт клиент, соединение переходит из режима двустороннего в режим исходящего копирования и клиент больше не может передавать сообщения CopyData. Аналогично, когда сообщение CopyDone передаёт сервер, соединение переходит в режим входящего копирования и сервер больше не может передавать сообщения CopyData. Когда сообщения CopyDone переданы обеими сторонами, режим копирования завершается и сервер возвращается в режим обработки команд. В случае обнаружения ошибки на стороне сервера в режиме двустороннего копирования, сервер выдаёт сообщение ErrorResponse, пропускает следующие сообщения клиента, пока не будет получено сообщение Sync, а затем выдаёт ReadyForQuery и возвращается к обычной обработке. Клиент должен воспринимать получение ErrorResponse как завершение двустороннего копирования; в этом случае сообщение CopyDone посылаться не должно. За дополнительной информацией о подразделе протокола, управляющем двусторонним копированием, обратитесь к Разделу 50.4.

Сообщения CopyInResponse, CopyOutResponse и CopyBothResponse содержат поля, из которых клиент может узнать количество столбцов в строке и код формата для каждого столбца. (В текущей реализации для всех столбцов в заданной операции `COPY` устанавливается один формат, но в конструкции сообщения это не заложено.)

50.2.7. Асинхронные операции

Возможны ситуации, в которых сервер будет отправлять клиенту сообщения, не предполагаемые потоком команд в текущем режиме. Клиенты должны быть готовы принять эти сообщения в любой момент, даже не в процессе выполнения запроса. Как минимум, следует проверять такие сообщения, прежде чем начинать читать ответ на запрос.

Сообщения NoticeResponse могут выдаваться вследствие внешней активности; например, если администратор инициирует «быстрое» отключение баз данных, сервер отправит NoticeResponse, сигнализирующее об этом факте, прежде чем закрывать соединение. Соответственно, клиенты должны быть готовы всегда принять и вывести сообщения NoticeResponse, даже когда соединение фактически простаивает.

Сообщения ParameterStatus будут выдаваться всякий раз, когда меняется действующее значение одного из параметров, об изменении которых, по мнению сервера, должен знать клиент. Чаще всего это происходит в ответ на SQL-команду `SET`, выполняемую клиентом и в таком случае это сообщение по сути синхронно — но состояние параметров может меняться и когда администратор изменяет файл конфигурации, а затем посылает серверу сигнал SIGHUP. Также, если действие команды `SET` отменяется, клиенту передаётся сообщение ParameterStatus, в котором отражается текущее значение параметра.

В настоящее время есть жёстко зафиксированный набор параметров, при изменении которых выдаётся ParameterStatus: `server_version`, `server_encoding`, `client_encoding`, `application_name`, `is_superuser`, `session_authorization`, `DateStyle`, `IntervalStyle`, `TimeZone`, `integer_datetimes` и `standard_conforming_strings`. (`server_encoding`, `TimeZone` и `integer_datetimes` не отслеживались до версии 8.0; `standard_conforming_strings` не отслеживался до 8.1; `IntervalStyle` — до версии 8.4; `application_name` — до версии 9.0.) Заметьте, что `server_version`, `server_encoding` и `integer_datetimes` — это псевдопараметры, которые не могут меняться после запуска сервера. Этот набор может быть изменён в будущем или даже будет настраиваемым. Соответственно, клиент может просто игнорировать сообщения ParameterStatus для параметров, которые ему неизвестны или не представляют интереса.

Если клиент выполняет команду `LISTEN`, сервер будет передавать ему сообщения NotificationResponse (не путайте с NoticeResponse!), когда для канала с тем же именем затем будет выполняться команда `NOTIFY`.

> **Примечание**
>
> В настоящее время, сообщение NotificationResponse может быть передано только вне транзакции, так что оно не может оказаться в середине серии ответов на команду, хотя может поступить сразу после ReadyForQuery. Однако полагаться на это при проектировании логики клиента неразумно. Лучше разработать код так, чтобы NotificationResponse могло быть принято в любой фазе протокола.

50.2.8. Отмена выполняющихся запросов

В процессе обработки запроса клиент может запросить отмену этого запроса. Запрос отмены не передаётся серверу непосредственно через открытое соединение из соображений эффективности: мы не хотим, чтобы сервер постоянно проверял поступающие от клиента сообщения в процессе выполнения запроса. Запросы отмены должны быть относительно редкими, поэтому мы выбрали для них не самый простой путь во избежание негативного влияния на обычный режим работы.

Чтобы запросить отмену запроса, клиент должен установить новое подключение к серверу и отправить ему сообщение CancelRequest, вместо StartupMessage, обычно передаваемого при установлении нового подключения. Сервер обработает полученную команду и закроет это подключение. По соображениям безопасности сервер не отвечает непосредственно на сообщение с запросом отмены.

Сообщение CancelRequest обрабатывается, только если оно содержит те же ключевые данные (PID и секретный ключ), что были переданы клиенту при запуске. Если PID и секретный ключ в запросе соответствуют данным выполняющегося в данный момент обслуживающего процесса, обработка текущего запроса в нём прерывается. (В существующей реализации это осуществляется путём передачи специального сигнала данному обслуживающему процессу.)

Сигнал отмены может подействовать, а может и не подействовать (например, если он поступит после того, как сервер завершит обработку запроса). Если отмена действительно происходит, текущая команда прерывается досрочно с сообщением об ошибке.

Вследствие такой организации, объясняемой соображениями безопасности и эффективности, клиент не может непосредственно определить, был ли запрос отмены успешен. Он должен продолжать ожидать ответа сервера на исходный запрос. Запрос отмены просто увеличивает вероятность того, что текущий запрос завершится быстрее, как и вероятность того, что он будет завершён не успешно, а с ошибкой.

Так как запрос отмены передаётся серверу через новое подключение к серверу, а не через обычный канал связи клиент-сервер, такие запросы могут выдавать любые процессы, а не только клиентский процесс, запрос которого требуется отменить. Это может дать дополнительную гибкость при построении многопроцессных приложений. Это также представляет собой угрозу безопасности, так как попытаться отменить запросы могут и неавторизованные пользователи. Для ликвидации этой угрозы в запросах отмены требуется передавать динамически генерируемый секретный ключ.

50.2.9. Завершение

Обычная процедура мягкого завершения заключается в том, что клиент отправляет сообщение Terminate и немедленно закрывает соединение. Получая это сообщение, обслуживающий процесс закрывает соединение и завершается.

В редких случаях (например, при отключении базы данных по команде администратора) обслуживающий процесс может отключиться, даже если клиент не запрашивает этого. В таких случаях, перед тем, как закрыть соединение, этот процесс пытается передать сообщение с ошибкой или уведомлением, в котором будет указана причина отключения.

Другие сценарии завершения возникают с различными вариантами отказа, например, при критическом сбое с дампом памяти на одной или другой стороне, при потере канала соединения,

потере синхронизации по границам сообщений и т. д. Если клиент или сервер обнаруживает, что соединение было неожиданно закрыто, он должен очистить ресурсы и завершиться. Клиент при этом может запустить новый обслуживающий процесс, переподключившись к серверу, если он сам хочет продолжать работу. Закрывать соединение также рекомендуется при получении сообщений нераспознанного типа, так как это может быть признаком потери синхронизации по границам сообщений.

При штатном или нештатном завершении сеанса любая открытая транзакция откатывается, а не фиксируется. Однако следует заметить, что при отключении клиента в процессе обработки запроса, отличного от `SELECT`, обслуживающий процесс вероятнее всего завершит запрос, прежде чем заметит отключение. Если запрос выполняется не в блоке транзакции (вне последовательности `BEGIN` ... `COMMIT`), его результаты могут быть зафиксированы до того, как будет обнаружено отключение.

50.2.10. Защита сеанса с SSL

Если Postgres Pro был собран с поддержкой SSL, взаимодействие клиента с сервером может быть зашифровано с применением SSL. Это обеспечивает защиту на уровне канала в среде, где злоумышленники могут перехватить трафик сеанса. За дополнительными сведениями о шифровании трафика сеансов Postgres Pro с использованием SSL обратитесь к Разделу 17.9.

Чтобы начать сеанс с SSL-шифрованием, клиент передаёт серверу вместо StartupMessage сообщение SSLRequest. В ответ сервер передаёт один байт, содержащий символ `S` или `N`, показывающий, что он желает, либо не желает включать SSL, соответственно. Если клиент не удовлетворён ответом, он должен закрыть соединение на этом этапе. Чтобы продолжить установление соединения после получения `S`, он выполняет начальное согласование SSL с сервером (не описывается здесь, так как относится к протоколу SSL). Если эта процедура выполняется успешно, он продолжает соединение, передавая обычное сообщение StartupMessage. При этом StartupMessage и все последующие данные будут защищены SSL-шифрованием. Чтобы продолжить после получения `N`, клиент может передать обычное сообщение StartupMessage и дальше взаимодействовать с сервером без шифрования.

Клиент также должен быть готов обработать сообщение ErrorMessage, полученное от сервера в ответ на SSLRequest. Такая ситуация возможна, только если сервер имеет версию, которая ещё не поддерживала SSL в Postgres Pro. (Такие серверы сейчас антикварная редкость, и скорее всего их уже не встретить в природе.) В этом случае соединение должно быть закрыто, но клиент может решить открыть новое соединение, не запрашивая SSL-шифрование.

Когда SSL-шифрование может быть включено, ожидается, что сервер передаст только один байт, содержащий символ `S`, а затем будет ожидать от клиента начала согласования SSL. Если на этом этапе доступны для чтения дополнительные байты, это может означать, что злоумышленник пытается выполнить атаку с переполнением буфера (*CVE-2021-23222*). Код на стороне клиента должен либо прочитать ровно один байт, а затем передать сокет библиотеке SSL, либо, если были прочитаны дополнительные байты, считать это нарушением протокола.

Начальный запрос SSLRequest может также передаваться при установлении соединения, открываемого для передачи сообщения CancelRequest.

Так как в самом протоколе не предусмотрено принудительное включение SSL-шифрования сервером, администратор может настроить сервер так, чтобы в качестве дополнительного условия при проверке подлинности клиента он не принимал незашифрованные сеансы.

50.3. Аутентификация SASL

SASL — это инфраструктура аутентификации для протоколов, ориентированных на соединения. На данный момент Postgres Pro реализует только один механизм SASL, SCRAM-SHA-256, но в будущем могут появиться и другие. Далее описывается, как в принципе осуществляется аутентификация SASL, а в следующем подразделе более подробно рассматривается SCRAM-SHA-256.

Поток сообщений аутентификации SASL

1. Чтобы начать обмен по схеме аутентификации SASL, сервер передаёт сообщение AuthenticationSASL. Оно содержит список механизмов аутентификации SASL, с которыми может работать сервер, в порядке предпочтений сервера.
2. Клиент выбирает один из поддерживаемых механизмов из списка и передаёт серверу сообщение SASLInitialResponse. Это сообщение содержит имя выбранного механизма и может содержать «Начальный ответ клиента», если это использует выбранный механизм.
3. За этим следует одно или нескольких сообщений вызова со стороны сервера и ответов со стороны клиента. Все вызовы сервер передаёт в сообщениях AuthenticationSASLContinue, а клиент отвечает на них сообщениями SASLResponse. Частные детали сообщений зависят от конкретного механизма.
4. Наконец, когда обмен аутентификационной информацией заканчивается успешно, сервер передаёт сообщение AuthenticationSASLFinal и сразу за ним сообщение AuthenticationOk. В сообщении AuthenticationSASLFinal передаются дополнительные данные от сервера клиенту, содержимое которых определяется выбранным механизмом аутентификации. Если механизм аутентификации не требует передавать дополнительные данные в завершение обмена, сообщение AuthenticationSASLFinal опускается.

В случае ошибки сервер может прервать процесс аутентификации на любом этапе и передать сообщение ErrorMessage.

50.3.1. Аутентификация SCRAM-SHA-256

Аутентификация *SCRAM-SHA-256* (далее называемая просто *SCRAM*) — единственный реализованный на данный момент механизм SASL. Она подробно описывается в RFC 7677 и RFC 5802.

Когда в Postgres Pro задействуется SCRAM-SHA-256, сервер игнорирует имя пользователя, которое клиент передаёт в `client-first-message`. Вместо этого используется имя, переданное ранее в стартовом сообщении. Согласно спецификации SCRAM, имя пользователя должно быть в UTF-8, но Postgres Pro поддерживает разные кодировки символов, и значит, имя пользователя Postgres Pro не всегда будет представимо в UTF-8.

В спецификации SCRAM говорится, что пароль также должен передаваться в UTF-8 и обрабатываться алгоритмом *SASLprep*. Однако Postgres Pro не требует, чтобы пароль представлялся в UTF-8. Когда устанавливается пароль пользователя, он обрабатывается алгоритмом SASLprep как пароль в UTF-8, вне зависимости от фактической кодировки. Однако если он представлен недопустимой для UTF-8 последовательностью байтов либо содержит комбинации байтов UTF-8, которые не принимает алгоритм SASLprep, это не будет считаться ошибкой — при аутентификации будет использоваться исходный пароль, без обработки SASLprep. Это позволяет нормализовать пароли, представленные в UTF-8, и при этом использовать пароли не в UTF-8, а также не требует, чтобы система знала, в какой кодировке задан пароль.

Связывание с каналом ещё не реализовано.

Пример

1. Сервер передаёт сообщение AuthenticationSASL. Оно содержит список механизмов аутентификации SASL, с которыми может работать сервер.
2. Клиент в ответ передаёт сообщение SASLInitialResponse, информирующее о выбранном механизме, `SCRAM-SHA-256`. В поле «Начального ответа клиента» это сообщение содержит данные SCRAM `client-first-message`.
3. Сервер передаёт сообщение AuthenticationSASLContinue, содержащее данные SCRAM `server-first-message`.
4. Клиент передаёт сообщение SASLResponse, содержащее данные SCRAM `client-final-message`.

5. Сервер передаёт сообщение AuthenticationSASLFinal, содержащее данные SCRAM `server-final-message`, и сразу за ним сообщение AuthenticationOk.

50.4. Протокол потоковой репликации

Чтобы инициировать потоковую репликацию, клиент передаёт в стартовом сообщении параметр `replication`. Логическое значение `true` этого параметра указывает обслуживающему процессу перейти в режим передачи WAL (walsender), в котором вместо SQL-операторов клиент может выдавать только ограниченный набор команд репликации. В режиме walsender можно использовать только протокол простых запросов. Команды репликации будут записываться в журнал сообщений сервера, если включён режим log_replication_commands. Если этот параметр имеет значение `database`, процесс walsender должен подключиться к базе данных, указанной в параметре `dbname`, что позволит использовать это подключение для логической репликации с указанной базой данных.

Для тестирования команд репликации вы можете установить соединение для репликации, запустив psql или другую программу на базе `libpq` со строкой подключения, включающей параметр `replication`, например так:

```
psql "dbname=postgres replication=database" -c "IDENTIFY_SYSTEM;"
```

Однако часто полезнее использовать pg_receivewal (для физической репликации) или pg_recvlogical (для логической).

В режиме walsender принимаются следующие команды:

`IDENTIFY_SYSTEM`

Запрашивает идентификационные данные сервера. Сервер возвращает набор результатов с одной строкой, содержащей четыре поля:

`systemid` (`text`)

Уникальный идентификатор системы, идентифицирующий кластер. По нему можно определить, что базовая резервная копия, из которой инициализировался резервный сервер, получена из того же кластера.

`timeline` (`int4`)

Идентификатор текущей линии времени. Также полезен для того, чтобы убедиться, что резервный сервер согласован с главным.

`xlogpos` (`text`)

Текущее положение сохранённых данных в WAL. Позволяет узнать, с какой позиции в журнале предзаписи может начаться потоковая передача.

`dbname` (`text`)

Подключённая база данных или NULL.

`SHOW` *`имя`*

Запрашивает у сервера текущее значение параметра времени выполнения. Эта команда подобна SQL-команде SHOW.

`имя`

Имя параметра времени выполнения. Доступные параметры описаны в Главе 18.

`TIMELINE_HISTORY` *`лин_врем`*

Запрашивает с сервера файл истории для линии времени *`лин_врем`*. Сервер возвращает набор результатов в одной строке, содержащей два поля. Эти поля обозначены как имеющие типы `text` и `bytea`, но фактически они содержат просто байты, не в текстовой кодировке и без экранирования:

`filename` (`text`)

Имя файла с историей линии времени, например `00000002.history`.

`content` (`bytea`)

Содержимое файла с историей линией времени.

`CREATE_REPLICATION_SLOT` *`имя_слота`* [`TEMPORARY`] { `PHYSICAL` [`RESERVE_WAL`] | `LOGICAL` *`модуль_вывода`* [`EXPORT_SNAPSHOT` | `NOEXPORT_SNAPSHOT` | `USE_SNAPSHOT`] }

Создаёт слот физической или логической репликации. Слоты репликации описаны подробно в Подразделе 25.2.6.

`имя_слота`

Имя создаваемого слота. Заданное имя должно быть допустимым для слота репликации (см. Подраздел 25.2.6.1).

`модуль_вывода`

Имя модуля вывода, применяемого для логического декодирования (см. Раздел 46.6).

`TEMPORARY`

Это указание отмечает, что данный слот репликации является временным. Временные слоты не сохраняются на диске и автоматически удаляются при ошибке или завершении сеанса.

`RESERVE_WAL`

Указывает, что этот слот физической репликации резервирует WAL немедленно. Без этого указания WAL резервируется только при подключении клиента потоковой репликации.

`EXPORT_SNAPSHOT`
`NOEXPORT_SNAPSHOT`
`USE_SNAPSHOT`

Эти указания выбирают, что делать со снимком, создаваемым при инициализации логического слота. С указанием `EXPORT_SNAPSHOT`, подразумеваемым по умолчанию, этот снимок будет экспортироваться для использования в других сеансах. Это указание нельзя использовать внутри транзакции. С указанием `USE_SNAPSHOT` снимок будет использоваться для текущей транзакции, в которой выполняется команда. Это указание должно использоваться в транзакции, при этом команда `CREATE_REPLICATION_SLOT` должна быть первой в этой транзакции. Наконец, с `NOEXPORT_SNAPSHOT` снимок будет использоваться только для логического декодирования в обычном режиме, но ничего больше с ним делать нельзя.

В ответ на эту команду сервер передаст набор результатов с одной строкой, содержащей следующие поля:

`slot_name` (`text`)

Имя создаваемого слота репликации.

`consistent_point` (`text`)

Позиция в WAL, в которой слот достиг согласованного состояния. Это самая ранняя позиция, с которой может начаться трансляция через этот слот репликации.

`snapshot_name` (`text`)

Идентификатор снимка, экспортированного командой. Этот снимок действителен до тех пор, пока через это соединение не будет выполнена следующая команда или соединение не будет закрыто. Null, если созданный слот — физический.

`output_plugin` (`text`)

Имя модуля вывода, используемого созданным слотом репликации. Null, если созданный слот — физический.

`START_REPLICATION` [`SLOT` *`имя_слота`*] [`PHYSICAL`] *`XXX/XXX`* [`TIMELINE` *`лин_врем`*]

Указывает серверу начать потоковую передачу WAL, начиная с позиции *`XXX/XXX`* в WAL. Если указывается параметр `TIMELINE`, передача начинается на линии времени *`лин_врем`*, иначе выбирается текущая линия времени сервера. Сервер может вернуть в ответ ошибку, например, если запрошенный сегмент WAL уже потерян. Если проблем не возникает, сервер возвращает сообщение CopyBothResponse, а затем начинает передавать поток WAL клиенту.

Если в параметрах передаётся *`имя_слота`*, сервер будет отражать состояние репликации в этом слоте и отслеживать, какие сегменты, а если включён режим `hot_standby_feedback`, то и в каких транзакциях, всё ещё нужны этому резервному серверу.

Если клиент запрашивает не последнюю, но существующую в истории сервера линию времени, сервер будет передавать весь WAL на этой линии времени, начиная с запрошенной стартовой точки до момента, когда сервер переключился на другую линию времени. Если клиент запрашивает передачу с начальной позицией точно в конце старой линии времени, сервер немедленно отвечает CommandComplete, не переходя в режим COPY.

После передачи всех записей WAL на линии времени, не являющейся текущей, сервер завершает потоковую передачу, выходя из режима копирования. Когда клиент подтверждает завершение передачи, также выходя из режима копирования, сервер возвращает набор результатов в одной строке с двумя столбцами, сообщая таким образом о следующей линии времени в истории сервера. В первом столбце передаётся идентификатор следующей линии времени (типа `int8`), а во втором — позиция в WAL, в которой произошло переключение (типа `text`). Обычно в этой же позиции завершается передача потока WAL, но возможны исключения, когда сервер может передавать записи WAL из старой линии времени, которые он сам ещё не воспроизвёл до переключения. Наконец сервер передаёт сообщение CommandComplete, после чего он готов принять следующую команду.

Данные WAL передаются в серии сообщений CopyData. (Это позволяет перемежать их с другой информацией; в частности, сервер может передать сообщение ErrorResponse, если он столкнулся с проблемами, уже начав передачу потока.) Полезная нагрузка каждого сообщения CopyData от сервера к клиенту содержит данные в одном из следующих форматов:

XLogData (B) — данные журнала транзакций

Byte1('w')

Указывает, что в этом сообщении передаются данные WAL.

Int64

Начальная точка данных WAL в этом сообщении.

Int64

Текущее положение конца WAL на сервере.

Int64

Показания системных часов сервера в момент передачи, в микросекундах с полуночи 2000-01-01.

Byte*n*

Фрагмент потока данных WAL.

Одна запись WAL никогда не разделяется на два сообщения XLogData. Когда запись WAL пересекает границу страницы WAL, и таким образом от неё уже оказывается отделена

продолжающая запись, её можно разделить на сообщения по границе страницы. Другими словами, первая основная запись WAL и продолжающие её записи могут быть переданы в различных сообщениях XLogData.

Primary keepalive message (B) — Сообщение об активности ведущего

Byte1('k')

Указывает, что это сообщение об активности отправителя.

Int64

Текущее положение конца WAL на сервере.

Int64

Показания системных часов сервера в момент передачи, в микросекундах с полуночи 2000-01-01.

Byte1

Значение 1 означает, что клиент должен ответить на это сообщение как можно скорее, во избежание отключения по тайм-ауту. Со значением 0 это не требуется.

Принимающий процесс может передавать ответы отправителю в любое время, используя один из следующих форматов данных (также в полезной нагрузке сообщения CopyData):

Standby status update (F) — Обновление состояния резервного сервера

Byte1('r')

Указывает, что это сообщение передаёт обновлённое состояние получателя.

Int64

Положение следующего за последним байтом WAL, полученным и записанным на диск на резервном сервере.

Int64

Положение следующего за последним байтом WAL, сохранённым на диске на резервном сервере.

Int64

Положение следующего за последним байтом WAL, применённым на резервном сервере.

Int64

Показания системных часов клиента в момент передачи, в микросекундах с полуночи 2000-01-01.

Byte1

Если содержит 1, клиент запрашивает от сервера немедленный ответ на это сообщение. Так клиент может запросить отклик сервера и проверить, продолжает ли функционировать соединение.

Hot Standby feedback message (F) — Сообщение обратной связи горячего резерва

Byte1('h')

Указывает, что это сообщение обратной связи горячего резерва.

Int64

Показания системных часов клиента в момент передачи, в микросекундах с полуночи 2000-01-01.

Int32

Текущее глобальное значение xmin данного резервного сервера, не учитывающее catalog_xmin всех слотов репликации. Если и это значение, и следующее catalog_xmin, равны 0, это воспринимается как уведомление о том, что через данное подключение больше не будут передаваться сообщения обратной связи горячего резерва. Последующие ненулевые сообщения могут возобновить работу механизма обратной связи.

Int32

Эпоха глобального идентификатора транзакции xmin на резервном сервере.

Int32

Наименьшее значение catalog_xmin для всех слотов репликации на резервном сервере. Значение 0 показывает, что на резервном сервере нет catalog_xmin, либо обратная связь горячего резерва отключена.

Int32

Эпоха идентификатора транзакции catalog_xmin на резервном сервере.

START_REPLICATION SLOT *имя_слота* LOGICAL *XXX/XXX* [(*имя_параметра* [*значение_параметра*] [, ...])]

Указывает серверу начать потоковую передачу WAL для логической репликации, начиная с позиции *XXX/XXX* в WAL. Сервер может вернуть в ответ ошибку, например, если запрошенный сегмент WAL уже потерян. Если проблем не возникает, сервер возвращает сообщение CopyBothResponse, а затем начинает передавать поток WAL клиенту.

Данные, передаваемые внутри сообщений CopyBothResponse, имеют тот же формат, что описан для команды START_REPLICATION ... PHYSICAL.

Обработку выводимых данных для передачи выполняет модуль вывода, связанный с выбранным слотом.

SLOT *имя_слота*

Имя слота, из которого передаются изменения. Это имя является обязательным, оно должно соответствовать существующему логическому слоту репликации, созданному командой CREATE_REPLICATION_SLOT в режиме LOGICAL.

XXX/XXX

Позиция в WAL, с которой должна начаться передача.

имя_параметра

Имя параметра, передаваемого модулю логического декодирования для выбранного слота.

значение_параметра

Необязательное значение, в форме строковой константы, связываемое с указанным параметром.

DROP_REPLICATION_SLOT *имя_слота* [WAIT]

Удаляет слот репликации, что приводит к освобождению всех занятых им ресурсов на стороне сервера. Если слот представляет собой логический слот, созданный не в той базе данных, к которой подключён walsender, команда завершается ошибкой.

имя_слота

Имя слота, подлежащего удалению.

WAIT

С этим указанием команда будет ждать, пока активный слот не станет неактивным (по умолчанию в этом случае выдаётся ошибка).

BASE_BACKUP [LABEL *'метка'*] [PROGRESS] [FAST] [WAL] [NOWAIT] [MAX_RATE *скорость*] [TABLESPACE_MAP]

Указывает серверу начать потоковую передачу базовой копии. Система автоматически переходит в режим резервного копирования до начала передачи, и выходит из него после завершения копирования. Эта команда принимает следующие параметры:

LABEL *'метка'*

Устанавливает метку для резервной копии. Если метка не задана, по умолчанию устанавливается метка base backup. Для метки действуют те же правила применения кавычек, что и для стандартных строк SQL при включённым режиме standard_conforming_strings.

PROGRESS

Запрашивает информацию, необходимую для отслеживания прогресса операции. Сервер передаёт в ответ приблизительный размер в заголовке каждого табличного пространства, исходя из которого можно понять, насколько продвинулась передача потока. Для вычисления этого размера анализируются размеры всех файлов ещё до начала передачи, и это может негативно повлиять на производительность — в частности, может увеличиться задержка передачи первых данных. Так как файлы базы данных могут меняться во время резервного копирования, оценка размера не будет точной; размер базы может увеличиться или уменьшиться за время от вычисления этой оценки до передачи актуальных файлов.

FAST

Запрашивает быструю контрольную точку.

WAL

Включает в резервную копию необходимые сегменты WAL. При этом в подкаталог pg_wal архива базового каталога будут включены все файлы с начала до конца копирования.

NOWAIT

По умолчанию при копировании ожидается завершение архивации последнего требуемого сегмента WAL либо выдаётся предупреждение, если архивация журнала не включена. Указание NOWAIT отключает и ожидание, и предупреждение, так что обеспечение наличия требуемого журнала становится задачей клиента.

MAX_RATE *скорость*

Ограничивает (сдерживает) максимальный объём данных, передаваемый от сервера клиенту за единицу времени. Единица измерения этого параметра — килобайты в секунду. Если задаётся этот параметр, его значение должно быть равно нулю, либо должно находиться в диапазоне от 32 (килобайт/сек) до 1 Гбайта/сек (включая границы). Если передаётся ноль, либо параметр не задаётся, скорость передачи не ограничивается.

TABLESPACE_MAP

Включает информацию о символических ссылках, представленных в каталоге pg_tblspc, в файл tablespace_map. Файл карты табличных пространств содержит имена всех ссылок, содержащихся в каталоге pg_tblspc/, и полный путь для каждой ссылки.

Когда запускается копирование, сервер сначала передаёт два обычных набора результатов, за которыми следуют один или более результатов CopyResponse.

В первом обычном наборе результатов передаётся начальная позиция резервной копии, в одной строке с двумя столбцами. В первом столбце содержится стартовая позиция в формате XLogRecPtr, а во втором идентификатор соответствующей линии времени.

Во втором обычном наборе результатов передаётся по одной строке для каждого табличного пространства. Эта строка содержит следующие поля:

`spcoid` (`oid`)

OID табличного пространства либо NULL, если это базовый каталог.

`spclocation` (`text`)

Полный путь к каталогу табличного пространства либо NULL, если это базовый каталог.

`size` (`int8`)

Приблизительный размер табличного пространства, если была запрошена информация о прогрессе операции; в противном случае NULL.

За вторым обычным набором результатов следует одна или несколько серий результатов CopyResponse, одна для основного каталога данных и по одной для каждого табличного пространства, отличного от `pg_default` и `pg_global`. Данные в CopyResponse представляют собой выгруженное в формате tar («формате обмена ustar», описанном в стандарте POSIX 1003.1-2008) содержимое табличных пространств, за исключением того, что два замыкающих блока нулей, описанных в стандарте, не передаются. После завершения передачи данных tar передаётся заключительный обычный набор результатов, в котором сообщается конечная позиция копии в WAL, в том же формате, что и стартовая позиция.

Архив tar каталога данных и всех табличных пространств будет содержать все файлы в этих каталогах, будь то файлы Postgres Pro или посторонние файлы, добавленные в эти каталоги. Исключение составляют только следующие файлы:

- `postmaster.pid`
- `postmaster.opts`
- различные временные файлы и каталоги, создаваемые в процессе работы сервером Postgres Pro, в частности файлы и каталоги с именами, начинающимися с `pgsql_tmp`
- `pg_wal`, включая подкаталоги. Если в резервную копию включаются файлы WAL, в архив входит преобразованная версия `pg_wal`, в которой будут находиться только файлы, необходимые для восстановления копии, но не всё остальное содержимое этого каталога
- `pg_dynshmem`, `pg_notify`, `pg_replslot`, `pg_serial`, `pg_snapshots`, `pg_stat_tmp` и `pg_subtrans` копируются как пустые каталоги (даже если это символические ссылки)
- файлы, кроме обычных файлов и каталогов, например, символические ссылки (кроме вышеупомянутых каталогов) и файлы специальных устройств, пропускаются (символические ссылки в `pg_tblspc` сохраняются).

Если файловая система сервера поддерживает это, в архив включается информация о владельце, группе и режиме файла.

50.5. Протокол логической потоковой репликации

В этом разделе описывается протокол логической репликации, регламентирующий поток сообщений, который запускается командой репликации `START_REPLICATION SLOT` *`имя_слота`* `LOGICAL`.

Протокол логической потоковой репликации построен на примитивах протокола физической потоковой репликации.

50.5.1. Параметры протокола логической потоковой репликации

Команда логической репликации `START_REPLICATION` принимает следующие параметры:

proto_version

Версия протокола. В настоящее время поддерживается только версия `1`.

publication_names

Список разделённых запятыми имён публикаций, на которые подписывается клиент (будет получать их изменения). Имена отдельных публикаций обрабатываются как стандартные имена объектов и могут так же заключаться в кавычки при необходимости.

50.5.2. Сообщения протокола логической репликации

Отдельные сообщения протокола рассматриваются в следующих подразделах. Собственно сообщения описаны в Раздел 50.9.

Все сообщения верхнего уровня начинаются с байта, определяющего тип сообщения. Хотя он представлен в коде символьным типом, это знаковый байт без явно заданной кодировки.

Так как в протоколе потоковой репликации передаётся длина сообщения, нет необходимости указывать длину в заголовках сообщений верхнего уровня.

50.5.3. Поток сообщений протокола логической репликации

За исключением команды `START_REPLICATION` и сообщений о прогрессе воспроизведения, весь информационный поток направлен от сервера к клиенту.

Протокол логической репликации передаёт отдельные транзакции одну за другой. Это значит, что все сообщения между парой сообщений Begin и Commit относятся к одной транзакции.

Каждая передаваемая транзакция содержит ноль или более сообщений DML (Insert, Update, Delete). В каскадной схеме она может также содержать сообщения Origin. Это сообщение показывает, что транзакция пришла с другого узла в схеме репликации. Так как этим узлом в контексте протокола логической репликации может быть что угодно, единственным идентификатором является его имя. Как воспринимать это имя (если это вообще нужно), определяют нижестоящие узлы. Сообщение Origin всегда передаётся перед всеми остальными сообщениями DML в транзакции.

Каждое DML-сообщение содержит идентификатор (OID) отношения, который указывает на целевое отношение на стороне публикации. Перед первым DML-сообщением для данного OID отношения будет отправлено сообщение Relation, описывающее схему данного отношения. Впоследствии будет отправлено новое сообщение Relation, если определение отношения изменится после отправки последнего сообщения Relation. (В протоколе предполагается, что клиент сможет кешировать метаданные для достаточно большого числа отношений.)

В сообщениях Relation типы столбцов определяются идентификаторами (OID). Для встроенного типа предполагается, что клиент может разрешить этот OID локально, поэтому никакие дополнительные данные не нужны. Однако для любых других типов перед сообщением Relation будет передаваться сообщение Type, связывающее имя типа с определённым OID. Таким образом, клиент, которому нужно однозначно определять типы столбцов в отношении, должен кешировать содержимое сообщений Type, и в случае обнаружения полученного OID в кеше использовать эту информацию, а в противном случае разрешать его локально.

50.6. Типы данных в сообщениях

В этом разделе описываются базовые типы данных, применяемые в сообщениях.

Int*n*(*i*)

Целое число из *n* бит с сетевым порядком байт (наиболее значащий байт первый). Если указано *i*, это поле будет содержать именно указанное значение, в противном случае значение переменное. Например: Int16, Int32(42).

Int*n*[*k*]

Массив из *k* *n*-битовых целых, каждое записывается с сетевым порядком байт. Длина массива *k* всегда определяется по предыдущему полю сообщения, например Int16[M].

String(*s*)

Строка, оканчивающаяся нулём (строка в стиле C). На длину строк ограничение не накладывается. Если указывается *s*, это поле будет содержать именно указанное значение, в противном случае значение переменное. Например: String, String("user").

> **Примечание**
>
> *Нет никакого предопределённого ограничения* длины строки, которую может возвратить сервер. Поэтому при реализации клиента лучше использовать расширяемый буфер, чтобы он мог принять строку любого размера, умещающуюся в памяти. Если такой возможности нет, прочитайте строку целиком и отбросьте последние символы, не помещающиеся в ваш буфер фиксированного размера.

Byte*n*(*c*)

В точности *n* байт. Если размер поля *n* задаётся не константой, он всегда определяется по предыдущему полю сообщения. Если указывается *c*, оно задаёт точное значение. Например: Byte2, Byte1('\n').

50.7. Форматы сообщений

В этом разделе подробно описывается формат каждого сообщения. Все сообщения помечены символами, обозначающими, какая сторона может их передавать: клиент (F), сервер (B) или обе стороны (F & B). Заметьте, что хотя каждое сообщение включает счётчик байт в начале, формат сообщения разработан так, чтобы конец сообщения можно было найти, не обращаясь к счётчику байт. Это помогает проверять корректность сообщений. (Исключением является сообщение CopyData, так как оно образует часть потока данных; содержимое любого отдельного сообщения CopyData нельзя интерпретировать само по себе.)

AuthenticationOk (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32(8)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(0)

Показывает, что проверка подлинности прошла успешно.

AuthenticationKerberosV5 (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32(8)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(2)

Указывает, что требуется проверка подлинности по протоколу Kerberos V5.

AuthenticationCleartextPassword (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32(8)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(3)

Указывает, что требуется пароль, передаваемый открытым текстом.

AuthenticationMD5Password (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32(12)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(5)

Указывает, что требуется пароль, преобразованный в хеш MD5.

Byte4

Значение соли, с которым должен хешироваться пароль.

AuthenticationSCMCredential (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32(8)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(6)

Указывает, что требуется сообщение с учётными данными SCM.

AuthenticationGSS (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32(8)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(7)

Указывает, что требуется проверка подлинности на базе GSSAPI.

AuthenticationSSPI (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32(8)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(9)

Указывает, что требуется проверка подлинности на базе SSPI.

AuthenticationGSSContinue (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(8)

Указывает, что это сообщение содержит данные GSSAPI или SSPI.

Byte*n*

Данные аутентификации для GSSAPI или SSPI.

AuthenticationSASL (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(10)

Указывает, что требуется проверка подлинности на базе SASL.

Тело сообщения содержит список механизмов аутентификации SASL в порядке предпочтений сервера. За последним именем механизма аутентификации должен идти завершающий нулевой байт. Для каждого механизма передаётся:

String

Имя механизма аутентификации SASL.

AuthenticationSASLContinue (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(11)

Указывает, что это сообщение содержит данные вызова SASL.

Byte*n*

Данные SASL, специфичные для применяемого механизма SASL.

AuthenticationSASLFinal (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(12)

Указывает, что аутентификация SASL завершена.

Byte*n*

«Дополнительные данные» результата SASL, специфичные для применяемого механизма SASL.

BackendKeyData (B)

Byte1('K')

Указывает, что это сообщение содержит ключевые данные для отмены запросов. Клиент должен сохранить эти данные, если ему нужна возможность впоследствии выдавать сообщения CancelRequest.

Int32(12)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32

PID обслуживающего процесса.

Int32

Секретный ключ обслуживающего процесса.

Bind (F)

Byte1('B')

Указывает, что это сообщение представляет команду Bind.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

String

Имя целевого портала (пустая строка выбирает безымянный портал).

String

Имя исходного подготовленного оператора (пустая строка выбирает безымянный подготовленный оператор).

Int16

Количество кодов форматов следующих параметров (обозначается ниже символом *C*). Может быть нулевым, что показывает, что параметры отсутствуют или все параметры передаются в формате по умолчанию (текстовом); либо равняться одному, в этом случае указанный один код формата применяется ко всем параметрам; либо может равняться действительному количеству параметров.

Int16[*C*]

Коды форматов параметров. В настоящее время допускаются коды ноль (текстовый формат) и один (двоичный).

Int16

Количество следующих значений параметров (может быть нулевым). Оно должно совпадать с количеством параметров, требующихся для запроса.

Затем для каждого параметра идёт следующая пара полей:

Int32

Длина значения параметра, в байтах (само поле длины не считается). Может быть нулевой. В качестве особого значения, -1 представляет значение NULL. В случае с NULL никакие байты значений далее не следуют.

Byte*n*

Значение параметра в формате, определённом соответствующим кодом формата. Переменная *n* задаёт длину значения.

За последним параметром идут следующие поля:

Int16

Количество кодов формата для следующих столбцов результата (обозначается ниже символом *R*). Может быть нулевым, что показывает, что столбцы результата отсутствуют или для всех столбцов должен использоваться формат по умолчанию (текстовый), либо равняться одному, в этом случае указанный один код формата применяется ко всем столбцам (если они есть), либо может равняться действительному количеству столбцов результата запроса.

Int16[*R*]

Коды форматов столбцов результата. В настоящее время допускаются коды ноль (текстовый формат) и один (двоичный).

BindComplete (B)

Byte1('2')

Указывает, что это сообщение, сигнализирующее о завершении Bind.

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

CancelRequest (F)

Int32(16)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(80877102)

Код запроса отмены. Это специально выбранное значение содержит `1234` в старших 16 битах и `5678` в младших 16 битах. (Во избежание неоднозначности этот код не должен совпадать с номером версии протокола.)

Int32

PID целевого обслуживающего процесса.

Int32

Секретный ключ целевого обслуживающего процесса.

Close (F)

Byte1('C')

Указывает, что это сообщение представляет команду Close.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Byte1

'`S`' для закрытия подготовленного оператора, '`P`' для закрытия портала.

String

Имя подготовленного оператора или портала, который должен быть закрыт (пустая строка выбирает безымянный подготовленный оператор или портал).

CloseComplete (B)

Byte1('3')

Указывает, что это сообщение, сигнализирующее о завершении Close.

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

CommandComplete (B)

Byte1('C')

Указывает, что это сообщение об успешном завершении команды.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

String

Тег команды. Обычно это одно слово, обозначающее завершённую команду SQL.

Для команды `INSERT` в качестве тега передаётся `INSERT` *`oid число_строк`*, где *`число_строк`* — количество вставленных строк. В поле *`oid`* передаётся идентификатор объекта вставленной строки, если *`число_строк`* равно 1 и в целевой таблице содержатся OID; в противном случае вместо *`oid`* передаётся 0.

Для команды `DELETE` в качестве тега передаётся `DELETE` *`число_строк`*, где *`число_строк`* — количество удалённых строк.

Для команды `UPDATE` в качестве тега передаётся `UPDATE` *`число_строк`*, где *`число_строк`* — количество изменённых строк.

Для команды `SELECT` или `CREATE TABLE AS` в качестве тега передаётся `SELECT` *`число_строк`*, где *`число_строк`* — число полученных строк.

Для команды `MOVE` в качестве тега передаётся `MOVE` *`число_строк`*, где *`число_строк`* — количество строк, на которое изменилась позиция курсора.

Для команды `FETCH` в качестве тега передаётся `FETCH` *`число_строк`*, где *`число_строк`* — количество строк, полученное через курсор.

Для команды `COPY` в качестве тега передаётся `COPY` *`число_строк`*, где *`число_строк`* — количество скопированных строк. (Заметьте: число строк выводится, начиная только с PostgreSQL 8.2.)

CopyData (F & B)

Byte1('d')

Указывает, что в этом сообщении передаются данные `COPY`.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Byte*n*

Данные, образующие часть информационного потока `COPY`. Сообщения от сервера всегда соответствуют отдельным строкам данных, но сообщения, передаваемые клиентами, могут разделять поток произвольным образом.

CopyDone (F & B)

Byte1('c')

Указывает, что это сообщение, сигнализирующее о завершении `COPY`.

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

CopyFail (F)

Byte1('f')

Указывает, что это сообщение, сигнализирующее об ошибке операции `COPY`.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

String

Сообщение об ошибке, описывающее причину сбоя операции.

CopyInResponse (B)

Byte1('G')

Указывает, что это сообщение является ответом на запуск входящего копирования. Получив его, клиент начинает передавать данные на вход операции копирования (если клиент не готов к этому, он передаёт сообщение CopyFail).

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int8

Значение 0 указывает, что для всей операции `COPY` применяется текстовый формат (строки разделяются символами новой строки, столбцы разделяются символами-разделителями и т. д.). Значение 1 указывает, что для всей операции копирования применяется двоичный формат (подобный формату DataRow). За дополнительными сведениями обратитесь к COPY.

Int16

Количество столбцов в копируемых данных (ниже обозначается символом *N*).

Int16[*N*]

Коды формата для каждого столбца. В настоящее время допускаются коды ноль (текстовый формат) и один (двоичный). Если общий формат копирования — текстовый, все эти коды должны быть нулевыми.

CopyOutResponse (B)

Byte1('H')

Указывает, что это сообщение является ответом на запуск исходящего копирования. За этим сообщением следуют данные, исходящие со стороны сервера.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int8

Значение 0 указывает, что для всей операции `COPY` применяется текстовый формат (строки разделяются символами новой строки, столбцы разделяются символами-разделителями и т. д.). Значение 1 указывает, что для всей операции копирования применяется двоичный формат (подобный формату DataRow). За дополнительными сведениями обратитесь к COPY.

Int16

Количество столбцов в копируемых данных (ниже обозначается символом *N*).

Int16[*N*]

Коды формата для каждого столбца. В настоящее время допускаются коды ноль (текстовый формат) и один (двоичный). Если общий формат копирования — текстовый, все эти коды должны быть нулевыми.

CopyBothResponse (B)

Byte1('W')

Указывает, что это сообщение является ответом на запуск двустороннего копирования. Это сообщение используется только для потоковой репликации.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int8

Значение 0 указывает, что для всей операции `COPY` применяется текстовый формат (строки разделяются символами новой строки, столбцы разделяются символами-разделителями и т. д.). Значение 1 указывает, что для всей операции копирования применяется двоичный формат (подобный формату DataRow). За дополнительными сведениями обратитесь к COPY.

Int16

Количество столбцов в копируемых данных (ниже обозначается символом *N*).

Int16[*N*]

Коды формата для каждого столбца. В настоящее время допускаются коды ноль (текстовый формат) и один (двоичный). Если общий формат копирования — текстовый, все эти коды должны быть нулевыми.

DataRow (B)

Byte1('D')

Указывает, что в этом сообщении передаётся строка данных.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int16

Количество последующих значений столбцов (может быть нулевым).

Затем для каждого столбца идёт следующая пара полей:

Int32

Длина значения столбца, в байтах (само поле длины не считается). Может быть нулевой. В качестве особого значения, -1 представляет значение NULL. В случае с NULL никакие байты значений далее не следуют.

Byte*n*

Значение столбца в формате, определённом соответствующим кодом формата. Переменная *n* задаёт длину значения.

Describe (F)

Byte1('D')

Указывает, что это сообщение представляет команду Describe.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Byte1

'`S`' для получения описания подготовленного оператора, '`P`' — портала.

String

Имя подготовленного оператора или портала, описание которого запрашивается (пустая строка выбирает безымянный подготовленный оператор или портал).

EmptyQueryResponse (B)

Byte1('I')

Указывает, что это сообщение является ответом на пустую строку запроса. (Это сообщение заменяет CommandComplete.)

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

ErrorResponse (B)

Byte1('E')

Указывает, что это сообщение ошибки.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Тело сообщения состоит из одного или нескольких определённых полей, за которыми в качестве завершающего следует нулевой байт. Поля могут идти в любом порядке. Для каждого поля передаётся:

Byte1

Код, задающий тип поля; ноль обозначает конец сообщения, после которого ничего нет. Типы полей, определённые в настоящее время, перечислены в Разделе 50.8. Так как в будущем могут появиться другие типы полей, клиенты должны просто игнорировать поля нераспознанного типа.

String

Значение поля.

Execute (F)

Byte1('E')

Указывает, что это сообщение представляет команду Execute.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

String

Имя портала, подлежащего выполнению (пустая строка выбирает безымянный портал).

Int32

Максимальное число строк, которое должно быть возвращено, если портал содержит запрос, возвращающий строки (в противном случае игнорируется). Ноль означает «без ограничения».

Flush (F)

Byte1('H')

Указывает, что это сообщение представляет команду Flush.

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

FunctionCall (F)

Byte1('F')

Указывает, что это сообщение представляет вызов функции.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int32

Задаёт идентификатор объекта вызываемой функции.

Int16

Количество кодов форматов следующих аргументов (обозначается ниже символом *C*). Может быть нулевым, что показывает, что аргументы отсутствуют или все аргументы передаются в формате по умолчанию (текстовом); либо равняться одному, в этом случае указанный один код формата применяется ко всем аргументами, либо может равняться действительному количеству аргументов.

Int16[*C*]

Коды форматов аргументов. В настоящее время допускаются коды ноль (текстовый формат) и один (двоичный).

Int16

Задаёт число аргументов, передаваемых функции.

Затем для каждого аргумента идёт следующая пара полей:

Int32

Длина значения аргумента, в байтах (само поле длины не считается). Может быть нулевой. В качестве особого значения, -1 представляет значение NULL. В случае с NULL никакие байты значений далее не следуют.

Byte*n*

Значение аргумента, в формате, определённом соответствующим кодом формата. Переменная *n* задаёт длину значения.

За последним аргументом идут следующие поля:

Int16

Код формата результата функции. В настоящее время допускается код ноль (текстовый формат) и один (двоичный).

FunctionCallResponse (B)

Byte1('V')

Указывает, что в этом сообщении передаётся результат вызова функции.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int32

Длина значения результата функции, в байтах (само поле длины не считается). Может быть нулевой. В качестве особого значения, -1 представляет значение NULL. В случае с NULL никакие байты значения далее не следуют.

Byte*n*

Значение результата функции в формате, определённом соответствующим кодом формата. Переменная *n* задаёт длину значения.

GSSResponse (F)

Byte1('p')

Обозначает это сообщение как ответ GSSAPI или SSPI. Заметьте, что оно также применяется для ответов SASL и при аутентификации по паролю. Точный тип сообщения можно определить из контекста.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Byte$n$

Данные сообщения, специфичные для GSSAPI/SSPI.

NegotiateProtocolVersion (B)

Byte1('v')

Указывает, что это сообщение согласования версии протокола.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int32

Новейшая младшая версия протокола, поддерживаемая сервером, для запрошенной клиентом старшей версии.

Int32

Число параметров протокола, не принятых сервером.

Затем для параметров протокола, не принятых сервером, передаётся:

String

Имя параметра.

NoData (B)

Byte1('n')

Указывает, что это сообщение сигнализирует об отсутствии данных.

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

NoticeResponse (B)

Byte1('N')

Указывает, что это сообщение представляет замечание.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Тело сообщения состоит из одного или нескольких определённых полей, за которыми в качестве завершающего следует нулевой байт. Поля могут идти в любом порядке. Для каждого поля передаётся:

Byte1

Код, задающий тип поля; ноль обозначает конец сообщения, после которого ничего нет. Типы полей, определённые в настоящее время, перечислены в Разделе 50.8. Так как в

будущем могут появиться другие типы полей, клиенты должны просто игнорировать поля нераспознанного типа.

String

Значение поля.

NotificationResponse (B)

Byte1('A')

Указывает, что это сообщение представляет уведомление.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int32

PID обслуживающего процесса, отправляющего уведомление.

String

Имя канала, для которого было выдано уведомление.

String

Строка «сообщения», сопровождающего уведомление.

ParameterDescription (B)

Byte1('t')

Указывает, что это сообщение представляет описание параметра.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int16

Количество параметров для оператора (может быть нулевым).

Затем для каждого параметра передаётся:

Int32

Задаёт идентификатор объекта типа данных параметра.

ParameterStatus (B)

Byte1('S')

Указывает, что в этом сообщении передаётся состояние параметра времени выполнения.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

String

Имя параметра времени выполнения, состояние которого передаётся.

String

Текущее значение параметра.

Parse (F)

Byte1('P')

Указывает, что это сообщение представляет команду Parse.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

String

Имя целевого подготовленного оператора (пустая строка выбирает безымянный подготовленный оператор).

String

Строка запроса, которая должна быть разобрана.

Int16

Количество типов параметров (может быть нулевым). Заметьте, что это значение представляет не число параметров, которые могут фигурировать в строке запроса, а число параметров, для которых клиент хочет предопределить типы.

Затем для каждого параметра передаётся:

Int32

Задаёт идентификатор объекта типа данных параметра. Указание нулевого значения равносильно отсутствию указания типа.

ParseComplete (B)

Byte1('1')

Указывает, что это сообщение, сигнализирующее о завершении Parse.

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

PasswordMessage (F)

Byte1('p')

Обозначает это сообщение как ответ SASL. Заметьте, что оно также применяется для ответов GSSAPI, SSPI и SASL. Точный тип сообщения можно определить из контекста.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

String

Пароль (зашифрованный, если требуется).

PortalSuspended (B)

Byte1('s')

Указывает, что это сообщение сигнализирует о приостановке портала. Заметьте, что оно выдаётся только при достижении ограничения числа строк, заданного в сообщении Execute.

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

Query (F)

Byte1('Q')

Указывает, что это сообщение представляет простой запрос.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

String

Собственно строка запроса.

ReadyForQuery (B)

Byte1('Z')

Определяет тип сообщения. Сообщение ReadyForQuery передаётся, когда сервер готов к новому циклу запросов.

Int32(5)

Длина содержимого сообщения в байтах, включая само поле длины.

Byte1

Индикатор текущего состояния транзакции на сервере. Возможные значения: 'I', транзакция неактивна (вне блока транзакции), 'T' в блоке транзакции, либо 'E' в блоке прерванной транзакции (запросы не будут обрабатываться до завершения блока).

RowDescription (B)

Byte1('T')

Указывает, что это сообщение представляет описание строки.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int16

Задаёт количество полей в строке (может быть нулевым).

Затем для каждого поля передаётся:

String

Имя поля.

Int32

Если поле связано со столбцом определённой таблицы, идентификатор объекта этой таблицы; в противном случае — ноль.

Int16

Если поле связано со столбцом определённой таблицы, номер атрибута для этого столбца; в противном случае — ноль.

Int32

Идентификатор объекта типа данных поля.

Int16

Размер типа данных (см. `pg_type.typlen`). Заметьте, что отрицательные значения показывают, что тип имеет переменную длину.

Int32

Модификатор типа (см. `pg_attribute.atttypmod`). Смысл этого модификатора зависит от типа.

Int16

Код формата, используемого для поля. В настоящее время допускаются коды ноль (текстовый формат) и один (двоичный). В сообщении RowDescription, возвращаемом вариацией Describe для оператора, код формата ещё не известен и всегда будет нулевым.

SASLInitialResponse (F)

Byte1('p')

Обозначает это сообщение как начальный ответ SASL. Заметьте, что оно также применяется для ответов GSSAPI, SSPI и при аутентификации по паролю. Точный тип сообщения определяется из контекста.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

String

Имя механизма аутентификации SASL, выбранного клиентом.

Int32

Длина последующего сообщения «Начальный ответ клиента», специфичного для механизма SASL, или -1, если начального ответа нет.

Byte*n*

«Начальный ответ», специфичный для механизма SASL.

SASLResponse (F)

Byte1('p')

Обозначает это сообщение как ответ SASL. Заметьте, что оно также применяется для ответов GSSAPI, SSPI и при аутентификации по паролю. Точный тип сообщения можно определить из контекста.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Byte*n*

Данные сообщения, специфичные для механизма SASL.

SSLRequest (F)

Int32(8)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(80877103)

Код запроса SSL. Это специально выбранное значение содержит `1234` в старших 16 битах и `5679` в младших 16 битах. (Во избежание неоднозначности этот код не должен совпадать с номером версии протокола.)

StartupMessage (F)

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(196608)

Номер версии протокола. В старших 16 битах задаётся старший номер версии (3 для протокола, описываемого здесь). В младших 16 битах задаётся младший номер версии (0 для протокола, описываемого здесь).

За номером версии протокола следует одна или несколько пар из имени параметра и строки значения. За последней парой имя/значение должен следовать нулевой байт. Передаваться параметры могут в любом порядке. Обязательным является только параметр `user`, остальные могут отсутствовать. Каждый параметр задаётся так:

String

Имя параметра. В настоящее время принимаются имена:

`user`

Имя пользователя баз данных, с которым выполняется подключение. Является обязательным, значения по умолчанию нет.

`database`

База данных, к которой выполняется подключение. По умолчанию подставляется имя пользователя.

`options`

Аргументы командной строки для обслуживающего процесса. (Этот способ считается устаревшим; теперь следует устанавливать отдельные параметры времени выполнения.) Пробелы в этой строке воспринимаются как разделяющие аргументы, если перед ними нет обратной косой черты (`\`); чтобы представить обратную косую черту буквально, продублируйте её (`\\`).

`replication`

Используется для подключения в режиме потоковой репликации, в котором вместо операторов SQL может выполняться небольшой набор команд репликации. Допустимые значения: `true`, `false` (по умолчанию) и `database`. За подробностями обратитесь к Разделу 50.4.

В дополнение к ним могут задаваться и другие параметры. Имена параметров, начинающиеся с `_pq_.`, резервируются для использования в расширениях протокола, а остальные воспринимаются как параметры времени выполнения, передаваемые во время запуска серверному процессу. Такие параметры будут применяться при запуске серверного процесса (после разбора аргументов командной строки, если они есть) и будут действовать как параметры сеанса по умолчанию.

String

Значение параметра.

Sync (F)

Byte1('S')

Указывает, что это сообщение представляет команду Sync.

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

Terminate (F)

Byte1('X')

Указывает, что это сообщение завершает сеанс.

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

50.8. Поля сообщений с ошибками и замечаниями

В этом разделе описываются поля, которые могут содержаться в сообщениях ErrorResponse и NoticeResponse. Для каждого типа поля определён свой идентификационный маркер. Заметьте, что в сообщении может содержаться поле любого из этих типов, но не больше одного раза.

`S`

Важность: поле содержит `ERROR`, `FATAL` или `PANIC` (в сообщении об ошибке), либо `WARNING`, `NOTICE`, `DEBUG`, `INFO` или `LOG` (в сообщении с замечанием), либо переведённые значения (ОШИБКА, ВАЖНО, ПАНИКА, ПРЕДУПРЕЖДЕНИЕ, ЗАМЕЧАНИЕ, ОТЛАДКА, ИНФОРМАЦИЯ, СООБЩЕНИЕ, соответственно). Это поле присутствует всегда.

`V`

Важность: поле содержит `ERROR`, `FATAL` или `PANIC` (в сообщении об ошибке) либо `WARNING`, `NOTICE`, `DEBUG`, `INFO` или `LOG` (в сообщении с замечанием). Это поле подобно `S`, но его содержимое никогда не переводится. Присутствует только в сообщениях, выдаваемых Postgres Pro версии 9.6 и новее.

`C`

Код: код SQLSTATE выданной ошибки (см. Приложение А). Не переводится на другие языки, присутствует всегда.

`M`

Сообщение: основное сообщение об ошибке, предназначенное для человека. Должно быть точным, но кратким (обычно в одну строку). Присутствует всегда.

`D`

Необязательное дополнительное сообщение об ошибке, передающее более детальную информацию о проблеме. Может занимать несколько строк.

`H`

Подсказка: необязательное предложение решения проблемы. Оно должно отличаться от подробного описания тем, что предлагает совет (не обязательно подходящий во всех случаях), а не сухие факты. Может располагаться в нескольких строках.

`P`

Позиция: значение поля представляет целочисленное число в ASCII, указывающее на положение ошибки в исходной строке запроса. Первый символ находится в позиции 1, при этом позиции отсчитываются по символам, а не по байтам.

`p`

Внутренняя позиция: она определяется так же, как поле `P`, но отражает положение ошибки во внутренне сгенерированной команде, а не в строке, переданной клиентом. Вместе с этим полем всегда присутствует поле `q`.

`q`

Внутренний запрос: текст внутренне сгенерированной команды, в которой произошла ошибка. Это может быть, например, SQL-запрос, выполняемый функцией на PL/pgSQL.

`W`

Где: указывает на контекст, в котором произошла ошибка. В настоящее время включает трассировку стека вызовов текущей функции на процедурном языке и внутренне сгенерированных запросов. Записи трассировки разделяются по строкам, вначале последняя.

`s`

Имя схемы: если ошибка связана с некоторым объектом базы данных, это поле содержит имя схемы, к которой относится объект (если такая есть).

`t`

Имя таблицы: если ошибка связана с некоторой таблицей, это поле содержит имя таблицы. (Узнать имя схемы таблицы можно из соответствующего отдельного поля.)

`c`

Имя столбца: если ошибка связана с некоторым столбцом таблицы, это поле содержит имя столбца. (Идентифицировать таблицу можно, обратившись к полям, содержащим имя таблицы и схемы.)

`d`

Имя типа данных: если ошибка связана с некоторым типом данных, это поле содержит имя типа. (Узнать имя схемы типа можно из соответствующего поля.)

`n`

Имя ограничения: если ошибка связана с некоторым ограничением, это поле содержит имя ограничения. Чтобы узнать, к какой таблице или домену она относится, обратитесь к полям, описанным выше. (В данном контексте индексы считаются ограничениями, даже если они были созданы не с синтаксисом ограничений.)

`F`

Файл: имя файла с исходным кодом, в котором была обнаружена ошибка.

`L`

Строка: номер строки в исходном коде, в которой была обнаружена ошибка.

`R`

Программа: имя программы в исходном коде, в которой была обнаружена ошибка.

> **Примечание**
>
> Поля, содержащие имена схемы, таблицы, столбца, типа данных и ограничения, выдаются только для ограниченного числа типов ошибок; см. Приложение A. Клиенты не должны рассчитывать на то, что присутствие одного из полей обязательно влечёт присутствие другого поля. Системные источники ошибок устанавливают связь между ними, но пользовательские функции могут использовать эти поля по-другому. Подобным образом, клиенты не должны полагаться на то, что эти поля ссылаются на актуальные объекты в текущей базе данных.

Клиент отвечает за форматирование отображаемой информации в соответствии с его нуждами; в частности, он должен разбивать длинные строки, как требуется. Символы новой строки, встречающиеся в полях сообщения об ошибке, должны обрабатываться, как разрывы абзацев, а не строк.

50.9. Форматы сообщений логической репликации

В этом разделе подробно описывается формат каждого сообщения логической репликации. Эти сообщения или выдаются через SQL-интерфейс слота репликации или передаются процессом walsender. Когда их передаёт walsender, они помещаются внутрь WAL-сообщений протокола репликации, описанных в Разделе 50.4, и в общем следуют тому же потоку сообщений, что и сообщения физической репликации.

Begin

Byte1('B')

Указывает, что это начальное сообщение.

Int64

Окончательный LSN транзакции.

Int64

Время фиксации транзакции. Значение задаётся в микросекундах, прошедших с начала эпохи Postgres Pro (2000-01-01).

Int32

Идентификатор транзакции.

Commit

Byte1('C')

Указывает, что это сообщение о фиксации.

Int8

Флаги; в настоящее время не используются (поле должно содержать 0).

Int64

LSN записи фиксации.

Int64

Конечный LSN транзакции.

Int64

Время фиксации транзакции. Значение задаётся в микросекундах, прошедших с начала эпохи Postgres Pro (2000-01-01).

Origin

Byte1('O')

Указывает, что это сообщение об источнике.

Int64

LSN записи фиксации на сервере-источнике.

String

Имя источника.

Заметьте, что внутри одной транзакции может быть несколько сообщений Origin.

Relation

Byte1('R')

Указывает, что это сообщение об отношении.

Int32

Идентификатор отношения.

String

Пространство имён (пустая строка для `pg_catalog`).

String

Имя отношения.

Int8

Свойство идентификации реплики для отношения (то же, что и `relreplident` в `pg_class`).

Int16

Число столбцов.

Затем для каждого столбца идёт следующий блок сообщения:

Int8

Флаги столбца. В настоящее время это может быть 0 (флагов нет) или 1 (столбец помечается как часть ключа).

String

Имя столбца.

Int32

Идентификатор типа данных столбца.

Int32

Модификатор типа столбца (`atttypmod`).

Тип

Byte1('Y')

Указывает, что это сообщение о типе.

Int32

Идентификатор типа данных.

String

Пространство имён (пустая строка для `pg_catalog`).

String

Имя типа данных.

Insert

Byte1('I')

Указывает, что это сообщение о добавлении данных.

Int32

Идентификатор отношения, соответствующий идентификатору в сообщении об отношении.

Byte1('N')

Обозначает следующее сообщение TupleData как содержащее новый кортеж.

TupleData

Блок сообщения TupleData, представляющий содержимое нового кортежа.

Update

Byte1('U')

Указывает, что это сообщение об изменении данных.

Int32

Идентификатор отношения, соответствующий идентификатору в сообщении об отношении.

Byte1('K')

Указывает, что следующий блок TupleData содержит ключ. Это поле является необязательным и присутствует, только если изменение затронуло столбцы, являющиеся частью индекса REPLICA IDENTITY.

Byte1('O')

Указывает, что следующий блок TupleData содержит старый кортеж. Это поле является необязательным и присутствует, только если у таблицы, в которой произошло изменение, свойство REPLICA IDENTITY равно FULL.

TupleData

Блок сообщения TupleData, представляющий содержимое старого кортежа или первичного ключа. Присутствует, только если перед ним идёт признак 'O' или 'K'.

Byte1('N')

Обозначает следующее сообщение TupleData как содержащее новый кортеж.

TupleData

Блок сообщения TupleData, представляющий содержимое нового кортежа.

Сообщение Update может содержать либо блок 'K', либо блок 'O', либо ни один из них, но не оба сразу.

Delete

Byte1('D')

Указывает, что это сообщение об удалении данных.

Int32

Идентификатор отношения, соответствующий идентификатору в сообщении об отношении.

Byte1('K')

Указывает, что следующий блок TupleData содержит ключ. Это поле присутствует, если таблица, в которой произошло удаление, использует индекс в качестве REPLICA IDENTITY.

Byte1('O')

Указывает, что следующий блок TupleData содержит старый кортеж. Это поле присутствует, если у таблицы, в которой произошло удаление, свойство REPLICA IDENTITY равно FULL.

TupleData

Блок сообщения TupleData, представляющий содержимое старого кортежа или первичного ключа, в зависимости от предыдущего поля.

Сообщение Delete может содержать либо блок 'K', либо блок 'O', но не оба сразу.

Описанные выше сообщения имеют следующие общие блоки.

TupleData

Int16

Число столбцов.

Затем для каждого столбца следует один из следующих блоков для каждого столбца:

Byte1('n')

Обозначает данные как значение NULL.

Или

Byte1('u')

Обозначает неизменённое значение TOAST (само значение не передаётся).

Или

Byte1('t')

Обозначает данные как значение в текстовом формате.

Int32

Длина значения столбца.

Byte*n*

Значение столбца в текстовом формате. (В будущих выпусках могут поддерживаться и другие форматы.) Здесь *n* — заданная выше длина.

50.10. Сводка изменений по сравнению с протоколом версии 2.0

В этом разделе представлен краткий список изменений к сведению разработчиков, желающих модернизировать существующие клиентские библиотеки до протокола 3.0.

В начальном стартовом пакете вместо фиксированного формата применяется гибкий формат списка строк. Заметьте, что теперь сеансовые значения по умолчанию для параметров времени выполнения можно задать непосредственно в стартовом пакете. (Вообще, это можно было делать и раньше, используя поле `options`, но из-за ограниченного размера `options` и невозможности задавать значения с пробелами, это вариант был не очень безопасным.)

Во всех сообщениях непосредственно за байтом типа сообщения следует счётчик длины (за исключением стартовых пакетов, в которых нет байта типа). Также заметьте, что байт типа теперь есть в сообщении PasswordMessage.

Сообщения ErrorResponse и NoticeResponse ('`E`' и '`N`') могут содержать несколько полей, из которых клиентский код может собрать сообщение об ошибке желаемого уровня детализации. Заметьте, что текст отдельных полей обычно не завершается новой строкой, тогда как в старом протоколе одиночная строка всегда завершалась так.

Сообщение ReadyForQuery ('`Z`') включает индикатор статуса транзакции.

Различие между типами данных BinaryRow и DataRow ушло; один тип сообщений DataRow позволяет возвращать данные во всех форматах. Заметьте, что формат DataRow был изменён для упрощения его разбора. Также изменилось представление двоичных значений: оно больше не привязано к внутреннему представлению сервера.

В протоколе появился новый подраздел «расширенный запрос», в котором добавлены типы сообщений для команд Parse, Bind, Execute, Describe, Close, Flush и Sync, а также типы серверных сообщений ParseComplete, BindComplete, PortalSuspended, ParameterDescription, NoData и CloseComplete. Существующие клиенты могут не подстраиваться под этот раздел протокола, но если они задействует его, это позволит улучшить производительность или функциональность.

Данные `COPY` теперь внедряются в сообщения CopyData и CopyDone. Есть чётко определённый способ восстановить работу в случае ошибок в процессе `COPY`. Специальная последняя строка «`\.`» больше не нужна, она не передаётся при выполнении `COPY OUT`. (Она по-прежнему воспринимается как завершающая последовательность в потоке `COPY IN`, но это считается устаревшим способом завершения, и в конце концов он будет исключён.) Поддерживается `COPY` в двоичном режиме. Сообщения CopyInResponse и CopyOutResponse включают поля, определяющие число столбцов и формат каждого столбца.

Изменилась структура сообщений FunctionCall и FunctionCallResponse. Сообщение FunctionCall теперь позволяет передавать функциям аргументы NULL. Ещё в нём могут передаваться параметры и получаться результаты в текстовом или двоичном формате. Не осталось повода считать сообщение FunctionCall потенциально небезопасным, так как оно не даёт прямого доступа к внутренней презентации данных на сервере.

Сервер отправляет сообщения ParameterStatus ('S') при попытке подключения для всех параметров, которые он считает интересными для клиентской библиотеки. Как следствие, при любом изменении активного значения одного из этих параметров также выдаётся сообщение ParameterStatus.

Сообщение RowDescription ('T') содержит поля с OID таблицы и номером столбца для каждого столбца описываемой строки. В нём также передаётся код формата для каждого столбца.

Сервер более не выдаёт сообщение CursorResponse ('P').

В сообщении NotificationResponse ('A') добавилось ещё одно строковое поле, в котором может передаваться строка «сообщения» от отправителя события NOTIFY.

Раньше сообщение EmptyQueryResponse ('I') включало пустой строковый параметр; теперь он ликвидирован.

Глава 51. Написание обработчика процедурного языка

Все функции, написанные на языке, вызываемом не через текущий интерфейс «версии 1» для компилируемых языков (а именно, это функции на процедурных языках и функции, написанные на SQL) выполняются через *обработчик вызова* для заданного языка. Задача такого обработчика вызова — выполнить функцию должным образом, например, интерпретируя для этого её исходный текст. В этой главе в общих чертах рассказывается, как можно написать обработчик нового процедурного языка.

Обработчик вызова процедурного языка — это «обычная» функция, которая разрабатывается на компилируемом языке, таком как C, вызывается через интерфейс версии 1, и регистрируется в Postgres Pro как не принимающая аргументы и возвращающая тип `language_handler`. Этот специальный псевдотип помечает функцию как обработчик вызова и препятствует её вызову непосредственно из команд SQL. Более подробно соглашение о вызовах и динамическая загрузка кода на C описывается в Разделе 35.9.

Обработчик вызова вызывается так же, как и любая другая функция: он получает указатель на переменную `struct FunctionCallInfoData`, содержащую значения аргументов и информацию о вызываемой функции, и должен вернуть результат типа `Datum` (и, возможно, установить признак `isnull` в структуре `FunctionCallInfoData`, если нужно вернуть результат SQL NULL). Отличие обработчика вызова от обычной вызываемой функцией состоит в том, что поле `flinfo->fn_oid` структуры `FunctionCallInfoData` для него будет содержать OID вызываемой функции, а не самого обработчика. По этому OID обработчик вызова должен понять, какую функцию вызывать. Кроме того, список передаваемых аргументов для него формируется в соответствии с объявлением целевой функции, а не обработчика вызова.

Обработчик вызова сам должен выбрать запись функции из системного каталога `pg_proc` и проанализировать типы аргументов и результата вызываемой функции. Содержимое предложения `AS` команды `CREATE FUNCTION` для этой функции будет находиться в столбце `prosrc` строки в `pg_proc`. Обычно это исходный текст на процедурном языке, но в принципе это может быть и что-то другое, например, путь к файлу или иные данные, говорящие обработчику вызова, что именно делать.

Часто функция многократно вызывается в одном SQL-операторе. Чтобы в таких случаях избежать повторных обращений за информацией о вызываемой функции, обработчик вызова может воспользоваться полем `flinfo->fn_extra`. Изначально оно содержит `NULL`, но обработчик вызова может поместить в него указатель на требуемую информацию. При последующих вызовах, если поле `flinfo->fn_extra` будет отлично от `NULL`, им можно воспользоваться и пропустить шаг получения этой информации. Обработчик вызова должен позаботиться о том, чтобы указатель в `flinfo->fn_extra` указывал на блок памяти, который не будет освобождён раньше, чем завершится запрос (именно столько может существовать структура `FmgrInfo`). В качестве одного из вариантов, этого можно добиться, разместив дополнительные данные в контексте памяти, заданном в `flinfo->fn_mcxt`; срок жизни таких данных обычно совпадает со сроком жизни самой структуры `FmgrInfo`. С другой стороны, обработчик может выбрать и более долгоживущий контекст памяти с тем, чтобы кешировать определения функций и между запросами.

Когда функция на процедурном языке вызывается как триггер, ей не передаются аргументы обычным способом; вместо этого поле `context` в `FunctionCallInfoData` указывает на структуру `TriggerData`, тогда как при обычном вызове функции оно содержит `NULL`. Обработчик языка, в свою очередь, должен каким-либо образом предоставить эту информацию функциям на этом процедурном языке.

Шаблон обработчика процедурного языка, написанный на C, выглядит так:

```
#include "postgres.h"
#include "executor/spi.h"
#include "commands/trigger.h"
```

```
#include "fmgr.h"
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(plsample_call_handler);

Datum
plsample_call_handler(PG_FUNCTION_ARGS)
{
    Datum          retval;

    if (CALLED_AS_TRIGGER(fcinfo))
    {
        /*
         * Вызывается как триггерная процедура
         */
        TriggerData    *trigdata = (TriggerData *) fcinfo->context;

        retval = ...
    }
    else
    {
        /*
         * Вызывается как функция
         */

        retval = ...
    }

    return retval;
}
```

Чтобы завершить код обработчика, нужно добавить лишь несколько тысяч строк вместо многоточий.

Скомпилировав функцию-обработчик языка в загружаемый модуль (см. Подраздел 35.9.5), этот язык (plsample) можно зарегистрировать следующими командами:

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS 'имя_файла'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

Хотя обработчика вызова достаточно для создания простейшего процедурного языка, есть ещё две функции, которые можно реализовать дополнительно, чтобы пользоваться языком было удобнее: функция *проверки* и *обработчик внедрённого кода*. Функцию проверки можно реализовать, чтобы производить проверку синтаксиса языка во время CREATE FUNCTION. Если же реализован обработчик внедрённого кода, этот язык будет поддерживать выполнение анонимных блоков кода командой DO.

Если для процедурного языка предоставляется функция проверки, она должна быть объявлена как функция, принимающая один параметр типа `oid`. Результат функции проверки игнорируется, так что она обычно объявляется как возвращающая тип `void`. Эта функция будет вызываться в

конце выполнения команды CREATE FUNCTION, создающей или изменяющей функцию, написанную на процедурном языке. Переданный ей OID указывает на строку в pg_proc для этой функции. Функция проверки должна выбрать эту строку обычным образом и произвести все необходимые проверки. Прежде всего нужно вызвать CheckFunctionValidatorAccess(), чтобы отличить явные вызовы этой функции от происходящих при выполнении команды CREATE FUNCTION. Затем обычно проверяется, например, что типы аргументов и результата функции поддерживаются языком и что тело функции синтаксически правильно для данного языка. Если функция проверки заключает, что всё в порядке, она должна просто завершиться. Если же она обнаруживает ошибку, она должна сообщить о ней через обычный механизм ereport(). Выданная таким образом ошибка приведёт к откату транзакции, так что определение некорректной функции зафиксировано не будет.

Функции проверки обычно должны учитывать параметр check_function_bodies: если он отключён, то дорогостоящие или зависящие от контекста проверки содержимого функции выполнять не следует. Если язык подразумевает выполнение кода в процессе компиляции, проверяющая функция должна избегать проверок, которые влекут за собой такое выполнение. В частности, указанный параметр отключает утилита pg_dump, чтобы она могла загружать функции на процедурных языках, не заботясь о побочных эффектах или зависимостях содержимого функций от других объектов базы. (Вследствие этого требования, обработчик языка не должен полагать, что функция прошла полную проверку. Смысл существования функции проверки не в том, чтобы убрать эти проверки из обработчика вызова, а в том, чтобы немедленно уведомить пользователя об очевидных ошибках при выполнении CREATE FUNCTION.) Хотя выбор, что именно должно проверяться, по большому счёту остаётся за функцией проверки, заметьте, что основной код CREATE FUNCTION выполняет присваивания SET, связанные с функцией, только когда check_function_bodies включён. Таким образом, проверки, результаты которых могут зависеть от параметров GUC, определённо должны опускаться, когда check_function_bodies отключён, во избежание ложных ошибок при восстановлении базы из копии.

Если для процедурного языка предоставляется обработчик встроенного кода, он должен объявляться в виде функции, принимающей один параметр типа internal. Результат такого обработчика игнорируется, поэтому обычно он объявляется как возвращающий тип void. Обработчик встроенного кода будет вызываться при выполнении оператора DO с данным процедурным языком. В качестве параметра ему на самом деле передаётся указатель на структуру InlineCodeBlock, содержащую информацию о параметрах DO, в частности, текст выполняемого анонимного блока внедрённого кода.

Все подобные объявления функций, а также саму команду CREATE LANGUAGE, рекомендуется упаковывать в *расширение* так, чтобы для установки языка было достаточно простой команды CREATE EXTENSION. За информацией о разработке расширений обратитесь к Разделу 35.15.

Реализация процедурных языков, включённых в стандартный дистрибутив, может послужить хорошим примером при написании собственных обработчиков языков. Её вы можете найти в подкаталоге src/pl дерева исходного кода. Некоторые полезные детали также можно узнать на странице справки CREATE LANGUAGE.

Глава 52. Написание обёртки сторонних данных

Все операции со сторонней таблицей производятся через созданную для неё обёртку сторонних данных, представляющую собой набор подпрограмм, которые вызывает ядро сервера. Обёртка сторонних данных отвечает за получение данных из удалённого источника данных и передачу их исполнителю запросов Postgres Pro. Чтобы поддерживалось изменение данных в сторонних таблицах, эту операцию также должна выполнять обёртка. В данной главе освещается написание обёртки сторонних данных.

Реализация обёрток сторонних данных, включённых в стандартный дистрибутив, может послужить хорошим примером при написании собственных обёрток. Её вы можете найти в подкаталоге `contrib` дерева исходного кода. Некоторые полезные детали также можно узнать на странице справки CREATE FOREIGN DATA WRAPPER.

> **Примечание**
>
> В стандарте SQL описан интерфейс для написания обёрток сторонних данных, но Postgres Pro не реализует его, так как это потребовало бы больших усилий, а данный стандартизированный API всё равно не получил широкого распространения.

52.1. Функции обёрток сторонних данных

Автор FDW (Foreign Data Wrapper, Обёртки сторонних данных) должен реализовать функцию-обработчик и может дополнительно добавить функцию проверки. Обе функции должны быть написаны на компилируемом языке, таком как C, и использовать интерфейс версии 1. Подробнее соглашение о вызовах и динамическая загрузка кода на C описывается в Разделе 35.9.

Функция-обработчик просто возвращает структуру с указателями на реализующие подпрограммы, которые будут вызываться планировщиком, исполнителем и различными служебными командами. Основная часть разработки FDW заключается в написании этих реализующих подпрограмм. Функция-обработчик должна быть зарегистрирована в Postgres Pro как функция без аргументов, возвращающая специальный псевдотип `fdw_handler`. Реализующие подпрограммы представляют собой обычные функции на C, которые не видны и не могут вызываться на уровне SQL. Они описаны в Разделе 52.2.

Функция проверки отвечает за проверку параметров, передаваемых с командами `CREATE` и `ALTER` для этой обёртки сторонних данных, а также параметров сторонних серверов, сопоставлений пользователей и сторонних таблиц, доступных через эту обёртку. Эта функция должна быть зарегистрирована как принимающая два аргумента: текстовый массив, содержащий параметры для проверки, и OID, представляющий тип объекта, с которым связаны эти параметры (в виде OID системного каталога, в котором будет сохраняться объект: `ForeignDataWrapperRelationId`, `ForeignServerRelationId`, `UserMappingRelationId` или `ForeignTableRelationId`). Если функция проверки отсутствует, параметры не проверяются ни при создании, ни при изменении объекта.

52.2. Подпрограммы обёртки сторонних данных

Функция-обработчик FDW возвращает структуру `FdwRoutine` (выделенную с помощью palloc), содержащую указатели на подпрограммы, которые реализуют описанные ниже функции. Из всех функций обязательными являются только те, что касаются сканирования, а остальные могут отсутствовать.

Тип структуры `FdwRoutine` объявлен в *src/include/foreign/fdwapi.h*, там же можно узнать дополнительные подробности.

52.2.1. Подпрограммы FDW для сканирования сторонних таблиц

```
void
GetForeignRelSize (PlannerInfo *root,
                   RelOptInfo *baserel,
                   Oid foreigntableid);
```

Выдаёт оценку размера отношения для сторонней таблицы. Она вызывается в начале планирования запроса, в котором сканируется сторонняя таблица. В параметре `root` передаётся общая информация планировщика о запросе, в `baserel` — информация о данной таблице, а в `foreigntableid` — OID записи в `pg_class` для данной таблицы. (Значение `foreigntableid` можно получить и из структуры данных планировщика, но простоты ради оно передаётся явно.)

Эта функция должна записать в `baserel->rows` ожидаемое число строк, которое будет получено при сканировании таблицы, с учётом фильтра, заданного ограничением выборки. Изначально в `baserel->rows` содержится просто постоянная оценка по умолчанию, которую следует заменить, если это вообще возможно. Функция также может поменять значение `baserel->width`, если она может дать лучшую оценку среднего размера строки результата. (Начальное значение зависит от типов столбцов и от среднего размера значений в них, рассчитанного при последнем `ANALYZE`.) Также эта функция может изменить значение `baserel->tuples`, если она может дать лучшую оценку общего количества строк в сторонней таблице. (Начальное значение берётся из поля `pg_class.reltuples`, которое содержит общее количество строк, полученное при последнем `ANALYZE`.)

За дополнительными сведениями обратитесь к Разделу 52.4.

```
void
GetForeignPaths (PlannerInfo *root,
                 RelOptInfo *baserel,
                 Oid foreigntableid);
```

Формирует возможные пути доступа для сканирования сторонней таблицы. Эта функция вызывается при планировании запроса. Ей передаются те же параметры, что и функции `GetForeignRelSize`, которая к этому времени уже будет вызвана.

Эта функция должна выдать минимум один путь доступа (узел `ForeignPath`) для сканирования сторонней таблицы и должна вызвать `add_path`, чтобы добавить каждый такой путь в `baserel->pathlist`. Для формирования узлов `ForeignPath` рекомендуется вызывать `create_foreignscan_path`. Данная функция может выдавать несколько путей доступа, то есть путей, для которых по заданным `pathkeys` можно получить уже отсортированный результат. Каждый путь доступа должен содержать оценки стоимости и может содержать любую частную информацию FDW, необходимую для выбора целевого метода сканирования.

За дополнительными сведениями обратитесь к Разделу 52.4.

```
ForeignScan *
GetForeignPlan (PlannerInfo *root,
                RelOptInfo *baserel,
                Oid foreigntableid,
                ForeignPath *best_path,
                List *tlist,
                List *scan_clauses,
                Plan *outer_plan);
```

Создаёт узел плана `ForeignScan` из выбранного пути доступа к сторонней таблице. Эта функция вызывается в конце планирования запроса. Ей передаются те же параметры, что и `GetForeignRelSize`, плюс выбранный путь `ForeignPath` (до этого сформированный функциями `GetForeignPaths`, `GetForeignJoinPaths` или `GetForeignUpperPaths`), целевой список, который должен быть выдан этим узлом плана, условия ограничения, которые должны применяться для данного узла, и внешний вложенный подплан `ForeignScan`, применяемый для перепроверок,

выполняемых функцией RecheckForeignScan. (Если путь задаётся для соединения, а не для базового отношения, в foreigntableid передаётся InvalidOid.)

Эта функция должна создать и выдать узел плана ForeignScan; для формирования этого узла рекомендуется использовать make_foreignscan.

За дополнительными сведениями обратитесь к Разделу 52.4.

```
void
BeginForeignScan (ForeignScanState *node,
                  int eflags);
```

Начинает сканирование сторонней таблицы. Эта функция вызывается при запуске исполнителя. Она должна выполнить все подготовительные действия, необходимые для осуществления сканирования, но не должна собственно производить сканирование (оно должно начаться с первым вызовом IterateForeignScan). Узел ForeignScanState уже был создан, но его поле fdw_state по-прежнему NULL. Информацию о сканируемой таблице можно получить через узел ForeignScanState (в частности, из нижележащего узла ForeignScan, содержащего частную информацию FDW, заданную функцией GetForeignPlan). Параметр eflags содержит битовые флаги, описывающие режим работы исполнителя для этого узла плана.

Заметьте, что когда (eflags & EXEC_FLAG_EXPLAIN_ONLY) не равно нулю, эта функция не должна выполнять какие-либо внешне проявляющиеся действия; она должна сделать только то, что необходимо для получения состояния узла, подходящего для ExplainForeignScan и EndForeignScan.

```
TupleTableSlot *
IterateForeignScan (ForeignScanState *node);
```

Выбирает одну строку из стороннего источника и возвращает её в слоте таблицы кортежей (для этой цели следует использовать ScanTupleSlot, переданный с узлом). Когда строки заканчиваются, возвращает NULL. Инфраструктура слотов таблицы кортежей позволяет возвращать как физические, так и виртуальные кортежи; в большинстве случаев второй вариант предпочтительнее с точки зрения производительности. Заметьте, что эта функция вызывается в контексте кратковременной памяти, который будет сбрасываться между вызовами. Если вам нужна более долгоживущая память, создайте соответствующий контекст в BeginForeignScan либо используйте es_query_cxt из структуры EState, переданной с узлом.

Возвращаемые строки должны соответствовать целевому списку fdw_scan_tlist, если он передаётся, а в противном случае — типу строки сканируемой сторонней таблицы. Если вы решите для оптимизации не возвращать ненужные столбцы, в их позиции нужно вставить NULL, либо сформировать список fdw_scan_tlist без этих столбцов.

Заметьте, что для исполнителя Postgres Pro не важно, удовлетворяют ли возвращаемые строки каким-либо ограничениям, определённым для сторонней таблицы — но это важно для планировщика, так что запросы могут оптимизироваться некорректно, если в сторонней таблице будут видны строки, не удовлетворяющие объявленному ограничению. Если ограничение нарушается, тогда как пользователь объявил, что оно должно выполняться, может быть уместно сообщить об ошибке (точно так же, как и при несовпадении типов данных).

```
void
ReScanForeignScan (ForeignScanState *node);
```

Перезапускает сканирование с начала. Заметьте, что значения параметров, от которых зависит сканирование, могли измениться, так что новое сканирование не обязательно вернёт те же строки.

```
void
EndForeignScan (ForeignScanState *node);
```

Завершает сканирование и освобождает ресурсы. Обычно при этом не нужно освобождать память, выделенную через palloc, но например, открытые файлы и подключения к удалённым серверам следует закрыть.

52.2.2. Подпрограммы FDW для сканирования сторонних соединений

Если FDW поддерживает соединения на удалённой стороне (вместо того, чтобы считывать данные обеих таблиц и выполнять соединения локально), она должна предоставить эту реализующую подпрограмму:

```
void
GetForeignJoinPaths (PlannerInfo *root,
                     RelOptInfo *joinrel,
                     RelOptInfo *outerrel,
                     RelOptInfo *innerrel,
                     JoinType jointype,
                     JoinPathExtraData *extra);
```

Формирует возможные пути доступа для соединения двух (и более) сторонних таблиц, принадлежащих одному стороннему серверу. Эта необязательная функция вызывается во время планирования запроса. Как и `GetForeignPaths`, эта функция должна построить пути `ForeignPath` для переданного `joinrel` и вызвать `add_path`, чтобы добавить эти пути в набор путей, подходящих для соединения. Но, в отличие от `GetForeignPaths`, эта функция не обязательно должна возвращать минимум один путь, так как всегда возможен альтернативный путь с локальным соединением таблиц.

Заметьте, что эта функция будет вызываться неоднократно для одного и того же соединения с разными комбинациями внутреннего и внешнего отношений; минимизировать двойную работу должна сама FDW.

Если для соединения выбирается путь `ForeignPath`, он будет представлять весь процесс соединения; пути, сформированные для задействованных таблиц и подчинённых соединений, в нём применяться не будут. Далее этот путь соединения обрабатывается во многом так же, как и путь сканирования одной сторонней таблицы. Одно различие состоит в том, что `scanrelid` результирующего плана узла `ForeignScan` должно быть равно нулю, так как он не представляет какое-либо одно отношение; вместо этого набор соединяемых отношений представляется в поле `fs_relids` узла `ForeignScan`. (Это поле заполняется автоматически кодом ядра планировщика, так что FDW делать это не нужно.) Ещё одно отличие в том, что список столбцов для удалённого соединения нельзя получить из системных каталогов и поэтому FDW должна выдать в `fdw_scan_tlist` требуемый список узлов `TargetEntry`, представляющий набор столбцов, которые будут выдаваться во время выполнения в возвращаемых кортежах.

За дополнительными сведениями обратитесь к Разделу 52.4.

52.2.3. Подпрограммы FDW для планирования обработки после сканирования/соединения

Если FDW поддерживает удалённое выполнение операций после сканирования/соединения, например, удалённое агрегирование, она должна предоставить эту реализующую подпрограмму:

```
void
GetForeignUpperPaths (PlannerInfo *root,
                      UpperRelationKind stage,
                      RelOptInfo *input_rel,
                      RelOptInfo *output_rel);
```

Формирует возможные пути доступа для обработки *верхнего отношения*. Этот термин планировщика подразумевает любую обработку запросов после сканирования/соединения, в частности, агрегирование, вычисление оконных функций, сортировку и изменение таблиц. Эта необязательная функция вызывается во время планирования запроса. В настоящее время она вызывается, только если все базовые отношения, задействованные в запросе, относятся к одной FDW. Эта функция должна построить пути `ForeignPath` для любых действий после сканирования/

соединения, которые FDW умеет выполнять удалённо, и вызвать add_path, чтобы добавить эти пути к указанному верхнему отношению. Как и GetForeignJoinPaths, эта функция не обязательно должна возвращать какие-либо пути, так как всегда возможны пути с локальной обработкой.

Параметр stage определяет, какой шаг после сканирования/соединения рассматривается в данный момент. Параметр output_rel указывает на верхнее отношение, которое должно получить пути, представляющие вычисление этого шага, а input_rel — на отношение, представляющее входные данные для этого шага. (Заметьте, что пути ForeignPath, добавляемые в output_rel, обычно не будут напрямую зависеть от путей input_rel, так как ожидается, что они будут обрабатываться снаружи. Однако изучить пути, построенные для предыдущего шага обработки, может быть полезно для исключения лишних операций при планировании.)

За дополнительными сведениями обратитесь к Разделу 52.4.

52.2.4. Подпрограммы FDW для изменения данных в сторонних таблицах

Если FDW поддерживает запись в сторонние таблицы, она должна предоставить некоторые или все подпрограммы, реализующие следующие функции, в зависимости от потребностей и возможностей FDW:

```
void
AddForeignUpdateTargets (Query *parsetree,
                         RangeTblEntry *target_rte,
                         Relation target_relation);
```

Операции UPDATE и DELETE выполняются со строками, ранее выбранными функциями сканирования таблицы. FDW может потребоваться дополнительная информация, например, ID строки или значения столбцов первичного ключа, чтобы точно знать, какую именно строку нужно изменить или удалить. Для этого данная функция может добавить дополнительные скрытые или «отбросовые» целевые столбцы в список столбцов, которые должны быть получены из сторонней таблицы во время UPDATE или DELETE.

Для этого добавьте в parsetree->targetList элементы TargetEntry, содержащие выражения для дополнительных выбираемых значений. У каждой такой записи должен быть признак resjunk = true и должно быть отдельное собственное имя resname, по которому она будет идентифицироваться во время выполнения. Избегайте использования имён вида ctid*N*, wholerow или wholerow*N*, так как столбцы с такими именами может генерировать ядро системы. Если дополнительные выражения сложнее, чем просто переменные, их нужно пропустить через функцию eval_const_expressions прежде чем добавлять в целевой список.

Хотя эта функция вызывается во время планирования, передаваемая ей информация несколько отличается от той, что получают другие подпрограммы планирования. В parsetree передаётся дерево разбора команды UPDATE или DELETE, а параметры target_rte и target_relation описывают целевую стороннюю таблицу.

Если указатель AddForeignUpdateTargets равен NULL, дополнительные целевые выражения не добавляются. (Это делает невозможным реализацию операций DELETE, хотя операция UPDATE может быть всё же возможна, если FDW идентифицирует строки, полагаясь на то, что первичный ключ не меняется.)

```
List *
PlanForeignModify (PlannerInfo *root,
                   ModifyTable *plan,
                   Index resultRelation,
                   int subplan_index);
```

Выполняет любые дополнительные действия планирования, необходимые для добавления, изменения или удаления в сторонней таблице. Эта функция формирует частную информацию FDW, которая будет добавлена в узел плана ModifyTable, осуществляющий изменение. Эта информация

должна возвращаться в списке (List); она будет доставлена в функцию BeginForeignModify на стадии выполнения.

В root передаётся общая информация планировщика о запросе, а в plan — узел плана ModifyTable, заполненный, не считая поля fdwPrivLists. Параметр resultRelation указывает на целевую стороннюю таблицу по номеру в списке отношений, а subplan_index определяет целевое отношение в данном узле ModifyTable, начиная с нуля; воспользуйтесь этим индексом, обращаясь к plan->plans или другой вложенной структуре узла plan.

За дополнительными сведениями обратитесь к Разделу 52.4.

Если указатель PlanForeignModify равен NULL, дополнительные действия во время планирования не предпринимаются, и в качестве fdw_private в BeginForeignModify поступит NULL.

```
void
BeginForeignModify (ModifyTableState *mtstate,
                    ResultRelInfo *rinfo,
                    List *fdw_private,
                    int subplan_index,
                    int eflags);
```

Начинает выполнение операции изменения данных в сторонней таблице. Эта подпрограмма выполняется при запуске исполнителя. Она должна выполнять любые подготовительные действия, необходимые для того, чтобы собственно произвести изменения в таблице. Впоследствии для каждого кортежа, который будет вставляться, изменяться или удаляться, будет вызываться ExecForeignInsert, ExecForeignUpdate или ExecForeignDelete.

В параметре mtstate передаётся общее состояние выполняемого плана узла ModifyTable; через эту структуру доступны глобальные сведения о плане и состояние выполнения. В rinfo передаётся структура ResultRelInfo, описывающая целевую стороннюю таблицу. (Если FDW нужно сохранить частное состояние, необходимое для этой операции, она может воспользоваться полем ri_FdwState структуры ResultRelInfo.) В fdw_private передаются частные данные, если они были сформированы процедурой PlanForeignModify. Параметр subplan_index определяет целевое отношение в данном узле ModifyTable, а в eflags передаются битовые флаги, описывающие режим работы исполнителя для этого узла плана.

Заметьте, что когда (eflags & EXEC_FLAG_EXPLAIN_ONLY) не равно нулю, эта функция не должна выполнять какие-либо внешне проявляющиеся действия; она должна сделать только то, что необходимо для получения состояния узла, подходящего для ExplainForeignModify и EndForeignModify.

Если указатель на BeginForeignModify равен NULL, никакое действие при запуске исполнителя не выполняется.

```
TupleTableSlot *
ExecForeignInsert (EState *estate,
                   ResultRelInfo *rinfo,
                   TupleTableSlot *slot,
                   TupleTableSlot *planSlot);
```

Вставляет один кортеж в стороннюю таблицу. В estate передаётся глобальное состояние выполнения запроса, а в rinfo — структура ResultRelInfo, описывающая целевую стороннюю таблицу. Параметр slot содержит кортеж, который должен быть вставлен; он будет соответствовать определению типа строки сторонней таблицы. Параметр planSlot содержит кортеж, сформированный вложенным планом узла ModifyTable; он отличается от slot тем, что может содержать дополнительные «отбросовые» столбцы. (Значение planSlot обычно не очень интересно для операций INSERT, но оно представлено для полноты.)

Возвращаемым значением будет либо слот, содержащий данные, которые были фактически вставлены (они могут отличаться от переданных данных, например, в результате действий

триггеров), либо NULL, если никакая строка фактически не была вставлена (опять же, обычно в результате действий триггеров). Чтобы вернуть результат, также можно использовать передаваемый на вход `slot`.

Данные в возвращаемом слоте используются, только если запрос `INSERT` содержит предложение `RETURNING` или для сторонней таблицы определён триггер `AFTER ROW`. Триггерам нужны все столбцы, но для предложения `RETURNING` FDW может ради оптимизации не возвращать некоторые или все столбцы, в зависимости от его содержания. Так или иначе, какой-либо слот необходимо вернуть, чтобы отметить, что операция успешна, иначе возвращённое число строк будет неверным.

Если указатель на `ExecForeignInsert` равен `NULL`, вставить данные в стороннюю таблицу не удастся, в ответ будет выдаваться сообщение об ошибке.

```
TupleTableSlot *
ExecForeignUpdate (EState *estate,
                   ResultRelInfo *rinfo,
                   TupleTableSlot *slot,
                   TupleTableSlot *planSlot);
```

Изменяет один кортеж в сторонней таблице. В `estate` передаётся глобальное состояние выполнения запроса, а в `rinfo` — структура `ResultRelInfo`, описывающая целевую стороннюю таблицу. Параметр `slot` содержит новые данные для кортежа; он будет соответствовать определению типа строки сторонней таблицы. Параметр `planSlot` содержит кортеж, сформированный вложенным планом узла `ModifyTable`; он отличается от `slot` тем, что может содержать дополнительные «отбросовые» столбцы. В частности, в этом слоте можно получить любые отбросовые столбцы, запрошенные в `AddForeignUpdateTargets`.

Возвращаемым значением будет либо слот, содержащий строку в состоянии после изменения (её содержимое может отличаться от переданного, например, в результате действий триггеров), либо NULL, если никакая строка фактически не была изменена (опять же, обычно в результате действий триггеров). Чтобы вернуть результат, также можно использовать передаваемый на вход `slot`.

Данные в возвращаемом слоте используются, только если запрос `UPDATE` содержит предложение `RETURNING` или для сторонней таблицы определён триггер `AFTER ROW`. Триггерам нужны все столбцы, но для предложения `RETURNING` FDW может ради оптимизации не возвращать некоторые или все столбцы, в зависимости от его содержания. Так или иначе, какой-либо слот необходимо вернуть, чтобы отметить, что операция успешна, иначе возвращённое число строк будет неверным.

Если указатель на `ExecForeignUpdate` равен `NULL`, изменить данные в сторонней таблице не удастся, а в ответ будет выдаваться сообщение об ошибке.

```
TupleTableSlot *
ExecForeignDelete (EState *estate,
                   ResultRelInfo *rinfo,
                   TupleTableSlot *slot,
                   TupleTableSlot *planSlot);
```

Удаляет один кортеж из сторонней таблицы. В `estate` передаётся глобальное состояние выполнения запроса, а в `rinfo` — структура `ResultRelInfo`, описывающая целевую стороннюю таблицу. Параметр `slot` при вызове не содержит ничего полезного, но в эту структуру можно поместить возвращаемый кортеж. Параметр `planSlot` содержит кортеж, сформированный вложенным планом узла `ModifyTable`; в частности, в нём могут содержаться отбросовые столбцы, запрошенные в `AddForeignUpdateTargets`. Отбросовые столбцы необходимы, чтобы определить, какой именно кортеж удалять.

Возвращаемым значением будет либо слот, содержащий строку, которая была удалена, либо NULL, если не удалена никакая строка (обычно в результате действия триггеров). Для размещения возвращаемого кортежа можно использовать передаваемый на вход `slot`.

Данные в возвращаемом слоте используются, только если запрос `DELETE` содержит предложение `RETURNING` или для сторонней таблицы определён триггер `AFTER ROW`. Триггерам нужны все

столбцы, но для предложения RETURNING FDW может ради оптимизации не возвращать некоторые или все столбцы, в зависимости от его содержания. Так или иначе, какой-либо слот необходимо вернуть, чтобы отметить, что операция успешна, иначе возвращённое число строк будет неверным.

Если указатель на ExecForeignDelete равен NULL, удалить данные из сторонней таблицы не удастся, а в ответ будет выдаваться сообщение об ошибке.

```
void
EndForeignModify (EState *estate,
                  ResultRelInfo *rinfo);
```

Завершает изменение данных в таблице и освобождает ресурсы. Обычно при этом не нужно освобождать память, выделенную через palloc, но например, открытые файлы и подключения к удалённым серверам следует закрыть.

Если указатель на EndForeignModify равен NULL, никакое действие при завершении исполнителя не выполняется.

```
int
IsForeignRelUpdatable (Relation rel);
```

Сообщает, какие операции изменения данных поддерживает указанная сторонняя таблица. Возвращаемое значение должно быть битовой маской кодов событий, обозначающих операции, поддерживаемые таблицей, и заданных в перечислении CmdType; то есть, (1 << CMD_UPDATE) = 4 для UPDATE, (1 << CMD_INSERT) = 8 для INSERT и (1 << CMD_DELETE) = 16 для DELETE.

Если указатель на IsForeignRelUpdatable равен NULL, предполагается, что сторонние таблицы позволяют добавлять, изменять и удалять строки, если FDW предоставляет процедуры для функций ExecForeignInsert, ExecForeignUpdate или ExecForeignDelete, соответственно. Данная функция необходима, только если FDW поддерживает операции изменения для одних таблиц и не поддерживает для других. (Хотя для этого можно выдать ошибку в подпрограмме, выполняющей операцию, а не задействовать эту функцию. Однако данная функция позволяет корректно отражать поддержку изменений в представлениях information_schema.)

Некоторые операции добавления, изменений и удаления данных в сторонних таблицах можно соптимизировать, применив альтернативный набор интерфейсов. Обычные интерфейсы для операций добавления, изменения и удаления выбирают строки с удалённого сервера, а затем модифицируют их по одной. В некоторых случаях такой подход «строка за строкой» необходим, но он может быть не самым эффективным. Если есть возможность определить на стороннем сервере, какие строки должны модифицироваться, собственно не считывая их, и если никакие локальные структуры (локальные триггеры уровня строк или ограничения WITH CHECK OPTION из родительских представлений) на эту операцию не влияют, её можно организовать так, чтобы она выполнялась целиком на удалённом сервере. Это позволяют осуществить описанные ниже интерфейсы.

```
bool
PlanDirectModify (PlannerInfo *root,
                  ModifyTable *plan,
                  Index resultRelation,
                  int subplan_index);
```

Определяет, возможно ли безопасно выполнить прямую модификацию на удалённом сервере. Если да, возвращает true, произведя требуемые для этого операции планирования. В противном случае возвращает false. Эта необязательная функция вызывается во время планирования запроса. Если результат этой функции положительный, на стадии выполнения будут вызываться BeginDirectModify, IterateDirectModify и EndDirectModify. Иначе модификация таблиц будет осуществляться посредством функций изменения, описанных выше. Данная функция принимает те же параметры, что и PlanForeignModify.

Для осуществления прямой модификации на удалённом сервере эта функция должна подставить в целевой подплан узел ForeignScan, выполняющий прямую модификацию на удалённом сервере.

В поле `operation` структуры `ForeignScan` должно быть установлено соответствующее значение перечисления `CmdType`: то есть, `CMD_UPDATE` для `UPDATE`, `CMD_INSERT` для `INSERT` и `CMD_DELETE` для `DELETE`.

За дополнительными сведениями обратитесь к Разделу 52.4.

Если указатель на `PlanDirectModify` равен `NULL`, сервер не будет пытаться произвести прямую модификацию.

```
void
BeginDirectModify (ForeignScanState *node,
                   int eflags);
```

Подготавливает прямую модификацию на удалённом сервере. Эта функция вызывается при запуске исполнителя. Она должна выполнить все подготовительные действия, необходимые для осуществления прямой модификации (модификация должна начаться с первым вызовом `IterateDirectModify`). Узел `ForeignScanState` уже был создан, но его поле `fdw_state` по-прежнему NULL. Информацию о модифицируемой таблице можно получить через узел `ForeignScanState` (в частности, из нижележащего узла `ForeignScan`, содержащего частную информацию FDW, заданную функцией `PlanDirectModify`). Параметр `eflags` содержит битовые флаги, описывающие режим работы исполнителя для этого узла плана.

Заметьте, что когда `(eflags & EXEC_FLAG_EXPLAIN_ONLY)` не равно нулю, эта функция не должна выполнять какие-либо внешне проявляющиеся действия; она должна сделать только то, что необходимо для получения состояния узла, подходящего для `ExplainDirectModify` и `EndDirectModify`.

Если указатель на `BeginDirectModify` равен `NULL`, сервер не будет пытаться произвести прямую модификацию.

```
TupleTableSlot *
IterateDirectModify (ForeignScanState *node);
```

Когда в запросе `INSERT`, `UPDATE` или `DELETE` отсутствует предложение `RETURNING`, просто возвращает NULL после прямой модификации на удалённом сервере. Когда в запросе есть это предложение, выбирает одну строку результата с данными, требующимися для вычисления `RETURNING`, и возвращает её в слоте таблицы кортежей (для этой цели следует использовать `ScanTupleSlot`, переданный с узлом). Данные, которые были фактически добавлены, изменены или удалены, нужно сохранить в `es_result_relation_info->ri_projectReturning->pi_exprContext->ecxt_scantuple` в структуре `EState`, переданной с узлом. Возвращает NULL, если строк больше нет. Заметьте, что эта функция вызывается в контексте кратковременной памяти, который будет сбрасываться между вызовами. Если вам нужна более долгоживущая память, создайте соответствующий контекст в `BeginDirectModify` либо используйте `es_query_cxt` из переданной с узлом структуры `EState`.

Возвращаемые строки должны соответствовать целевому списку `fdw_scan_tlist`, если он передаётся, а в противном случае — типу строки изменяемой сторонней таблицы. Если вы решите для оптимизации не возвращать ненужные столбцы, не требующиеся для получения `RETURNING`, в их позиции нужно вставить NULL, либо сформировать список `fdw_scan_tlist` без этих столбцов.

Независимо от того, есть ли в запросе это предложение или нет, число строк, возвращаемых запросом, должно увеличиваться самой FDW. Когда этого предложения в запросе нет, FDW должна также увеличивать число строк для узла `ForeignScanState` в случае `EXPLAIN ANALYZE`.

Если указатель на `IterateDirectModify` равен `NULL`, сервер не будет пытаться произвести прямую модификацию.

```
void
EndDirectModify (ForeignScanState *node);
```

Очищает ресурсы после непосредственной модификации на удалённом сервере. Обычно при этом не нужно освобождать память, выделенную через palloc, но например, открытые файлы и подключения к удалённому серверу следует закрыть.

Если указатель на `EndDirectModify` равен `NULL`, сервер не будет пытаться произвести прямую модификацию.

52.2.5. Подпрограммы FDW для блокировки строк

Если FDW желает поддержать функцию *поздней блокировки строк* (описанную в Разделе 52.5), она должна предоставить следующие реализующие подпрограммы:

```
RowMarkType
GetForeignRowMarkType (RangeTblEntry *rte,
                       LockClauseStrength strength);
```

Сообщает, какой вариант пометки строк будет использоваться для сторонней таблицы. Здесь `rte` представляет узел `RangeTblEntry` для таблицы, а `strength` описывает силу блокировки, запрошенную соответствующим предложением `FOR UPDATE/SHARE`, если оно имеется. Результатом должно быть значение перечисления `RowMarkType`.

Эта функция вызывается в процессе планирования запроса для каждой сторонней таблицы, которая участвует в запросе `UPDATE`, `DELETE` или `SELECT FOR UPDATE/SHARE`, и не является целевой в запросе `UPDATE` или `DELETE`.

Если указатель `GetForeignRowMarkType` равен `NULL`, всегда выбирается вариант `ROW_MARK_COPY`. (Вследствие этого, функция `RefetchForeignRow` никогда не будет вызываться, так что и её задавать не нужно.)

За подробностями обратитесь к Разделу 52.5.

```
HeapTuple
RefetchForeignRow (EState *estate,
                   ExecRowMark *erm,
                   Datum rowid,
                   bool *updated);
```

Повторно считывает один кортеж из сторонней таблицы после блокировки, если она требуется. В `estate` передаётся глобальное состояние выполнения запроса. В `erm` передаётся структура `ExecRowMark`, описывающая целевую стороннюю таблицу и тип запрашиваемой блокировки (если требуется блокировка). Параметр `rowid` идентифицирует считываемый кортеж. Параметр `updated` используется как выходной.

Эта функция должна вернуть копию выбранного кортежа (размещённую в памяти palloc) или `NULL`, если получить блокировку строки не удаётся. Тип запрашиваемой блокировки строки определяется значением `erm->markType`, которое было до этого возвращено функцией `GetForeignRowMarkType`. (Вариант `ROW_MARK_REFERENCE` означает, что нужно просто повторно выбрать кортеж, не запрашивая никакую блокировку, а `ROW_MARK_COPY` никогда не поступает в эту подпрограмму.)

Кроме того, переменной `*updated` следует присвоить `true`, если была считана изменённая версия кортежа, а не версия, полученная ранее. (Если FDW не знает этого наверняка, рекомендуется всегда возвращать `true`.)

Заметьте, что по умолчанию в случае неудачи при попытке получить блокировку строки должна выдаваться ошибка; значение `NULL` может возвращаться, только если в `erm->waitPolicy` выбран вариант `SKIP LOCKED`.

В `rowid` передаётся значение `ctid`, полученное ранее для строки, которую нужно считать повторно. Хотя значение `rowid` передаётся в виде `Datum`, в настоящее время это может быть

только tid. Такой интерфейс функции выбран с расчётом на то, чтобы в будущем в качестве идентификаторов строк могли приниматься и другие типы данных.

Если указатель на RefetchForeignRow равен NULL, повторно выбрать данные не удастся, в ответ будет выдаваться сообщение об ошибке.

За подробностями обратитесь к Разделу 52.5.

```
bool
RecheckForeignScan (ForeignScanState *node, TupleTableSlot *slot);
```

Перепроверяет, соответствует ли по-прежнему ранее возвращённый кортеж применимым условиям сканирования и соединения, и возможно выдаёт изменённую версию кортежа. Для обёрток сторонних данных, которые не выносят соединение наружу, обычно удобнее присвоить этому указателю NULL и задать fdw_recheck_quals. Однако когда внешние соединения выносятся наружу, недостаточно повторно применить к результирующему кортежу проверки, относящиеся ко всем базовым таблицам, даже если присутствуют все атрибуты, так как невыполнение некоторого условия может приводить и к обнулению некоторых атрибутов, а не только исключению этого кортежа. RecheckForeignScan может перепроверить условия и возвратить true, если они по-прежнему выполняются, или false в противном случае, но также она может записать в переданный слот кортеж на замену предыдущему.

Чтобы вынести соединение наружу, обёртка сторонних данных обычно конструирует альтернативный план локального соединения, применяемый только для перепроверок; он становится внешним подпланом узла ForeignScan. Когда требуется перепроверка, может быть выполнен этот подплан и результирующий кортеж сохранён в слоте. Этот план может не быть эффективным, так как ни одна базовая таблица не выдаст больше одной строки; например, он может реализовывать все соединения в виде вложенных циклов. Для поиска подходящего локального пути соединения в существующих путях можно воспользоваться функцией GetExistingLocalJoinPath. Функция GetExistingLocalJoinPath ищет непараметризованный путь в списке путей заданного отношения соединения. (Если такой путь не находится, она возвращает NULL, и в этом случае обёртка сторонних данных может построить локальный путь сама или решить не создавать пути доступа для этого соединения.)

52.2.6. Подпрограммы FDW для EXPLAIN

```
void
ExplainForeignScan (ForeignScanState *node,
                    ExplainState *es);
```

Дополняет вывод EXPLAIN для сканирования сторонней таблицы. Эта функция может вызывать ExplainPropertyText и связанные функции и добавлять поля в вывод EXPLAIN. Поля флагов в es позволяют определить, что именно выводить, а для выдачи статистики времени выполнения в случае с EXPLAIN ANALYZE можно проанализировать состояние узла ForeignScanState.

Если указатель ExplainForeignScan равен NULL, никакая дополнительная информация при EXPLAIN не выводится.

```
void
ExplainForeignModify (ModifyTableState *mtstate,
                      ResultRelInfo *rinfo,
                      List *fdw_private,
                      int subplan_index,
                      struct ExplainState *es);
```

Дополняет вывод EXPLAIN для изменений в сторонней таблице. Эта функция может вызывать ExplainPropertyText и связанные функции и добавлять поля в вывод EXPLAIN. Поля флагов в es позволяют определить, что именно выводить, а для выдачи статистики времени выполнения в случае с EXPLAIN ANALYZE можно проанализировать состояние узла ModifyTableState. Первые четыре аргумента у этой функции те же, что и у BeginForeignModify.

Если указатель ExplainForeignModify равен NULL, никакая дополнительная информация при EXPLAIN не выводится.

```
void
ExplainDirectModify (ForeignScanState *node,
                     ExplainState *es);
```

Дополняет вывод EXPLAIN для прямой модификации данных на удалённом сервере. Эта функция может вызывать ExplainPropertyText и связанные функции и добавлять поля в вывод EXPLAIN. Поля флагов в es позволяют определить, что именно выводить, а для выдачи статистики времени выполнения в случае EXPLAIN ANALYZE можно проанализировать состояние узла ForeignScanState.

Если указатель ExplainDirectModify равен NULL, никакая дополнительная информация при EXPLAIN не выводится.

52.2.7. Подпрограммы FDW для ANALYZE

```
bool
AnalyzeForeignTable (Relation relation,
                     AcquireSampleRowsFunc *func,
                     BlockNumber *totalpages);
```

Эта функция вызывается, когда для сторонней таблицы выполняется ANALYZE. Если FDW может собрать статистику для этой сторонней таблицы, эта функция должна вернуть true и передать в *func* указатель на функцию, которая будет выдавать строки выборки из таблицы, а в *totalpages* ожидаемый размер таблицы в страницах. В противном случае эта функция должна вернуть false.

Если FDW не поддерживает сбор статистики ни для каких таблиц, в AnalyzeForeignTable можно установить значение NULL.

Функция выдачи выборки, если она предоставляется, должна иметь следующую сигнатуру:

```
int
AcquireSampleRowsFunc (Relation relation, int elevel,
                       HeapTuple *rows, int targrows,
                       double *totalrows,
                       double *totaldeadrows);
```

Она должна выбирать из таблицы максимум *targrows* строк и помещать их в переданный вызывающим кодом массив *rows*. Возвращать она должна фактическое число выбранных строк. Кроме того, эта функция должна сохранить общее количество актуальных и «мёртвых»строк в таблице в выходных параметрах *totalrows* и *totaldeadrows*, соответственно. (Если для данной FDW нет понятия «мёртвых» строк, в *totaldeadrows* нужно записать 0.)

52.2.8. Подпрограммы FDW для IMPORT FOREIGN SCHEMA

```
List *
ImportForeignSchema (ImportForeignSchemaStmt *stmt, Oid serverOid);
```

Получает список команд, создающих сторонние таблицы. Эта функция вызывается при выполнении команды IMPORT FOREIGN SCHEMA; ей передаётся дерево разбора этого оператора и OID целевого стороннего сервера. Она должна вернуть набор строк C, в каждой из которых должна содержаться команда CREATE FOREIGN TABLE. Эти строки будут разобраны и выполнены ядром сервера.

В структуре ImportForeignSchemaStmt поле remote_schema задаёт имя удалённой схемы, из которой импортируются таблицы. Поле list_type устанавливает, как фильтровать имена таблиц: вариант FDW_IMPORT_SCHEMA_ALL означает, что нужно импортировать все таблицы в удалённой схеме (в этом случае поле table_list пустое), FDW_IMPORT_SCHEMA_LIMIT_TO означает, что нужно импортировать только таблицы, перечисленные в table_list, и FDW_IMPORT_SCHEMA_EXCEPT означает, что нужно исключить таблицы, перечисленные в списке table_list. В поле options передаётся список параметров для процесса импорта. Значение этих параметров определяется

самой FDW. Например, у FDW может быть параметр, определяющий, нужно ли сохранять у импортируемых столбцов атрибут NOT NULL. Эти параметры могут не иметь ничего общего с параметрами, которые принимает FDW в качестве параметров объектов базы.

FDW может игнорировать поле local_schema в ImportForeignSchemaStmt, так как ядро сервера само вставит это имя в разобранные команды CREATE FOREIGN TABLE.

Также, FDW может не выполнять сама фильтрацию по полям list_type и table_list, так как ядро сервера автоматически пропустит все возвращённые команды для таблиц, исключённых по заданным критериям. Однако часто лучше сразу избежать лишней работы, не формируя команды для исключаемых таблиц. Для проверки, удовлетворяет ли фильтру заданное имя сторонней таблицы, может быть полезна функция IsImportableForeignTable().

Если FDW не поддерживает импорт определений таблиц, указателю ImportForeignSchema можно присвоить NULL.

52.2.9. Подпрограммы FDW для параллельного выполнения

Узел ForeignScan может, хотя это не требуется, поддерживать параллельное выполнение. Параллельный ForeignScan будет выполняться в нескольких процессах и должен возвращать одну строку только единожды. Для этого взаимодействующие процессы могут координировать свои действия через фиксированного размера блоки в динамической разделяемой памяти. Эта разделяемая память не будет гарантированно отображаться по одному адресу в разных процессах, так что она не может содержать указатели. Все следующие функции являются необязательными, но большинство из них необходимы при реализации поддержки параллельного выполнения.

```
bool
IsForeignScanParallelSafe(PlannerInfo *root, RelOptInfo *rel,
                          RangeTblEntry *rte);
```

Проверяет, будет ли сканирование выполняться параллельным исполнителем. Эта функция будет вызываться, только когда планировщик считает, что параллельный план принципиально возможен, и должна возвращать true, если такое сканирование может безопасно выполняться параллельным исполнителем. Обычно это не так, если удалённый источник данных является транзакционным. Но возможно исключение, когда в подключении рабочего процесса к этому источнику каким-то образом используется тот же транзакционный контекст, что и в ведущем процессе.

Если эта функция не определена, считается, что сканирование должно происходить в ведущем процессе. Заметьте, что возвращённое значение true не означает, что само сканирование может выполняться в параллельном режиме, а только то, что сканирование будет производится в параллельном исполнителе. Таким образом, может быть полезно определить этот обработчик, даже если параллельное выполнение не поддерживается.

```
Size
EstimateDSMForeignScan(ForeignScanState *node, ParallelContext *pcxt);
```

Оценивает объём динамической разделяемой памяти, которая потребуется для параллельной операции. Это значение может превышать объём, который будет занят фактически, но не должно быть меньше. Возвращаемое значение задаётся в байтах. Эта функция является необязательной и может быть опущена, если не требуется; но в этом случае должны быть также опущены следующие три функции, так как для FDW не будет выделена разделяемая память.

```
void
InitializeDSMForeignScan(ForeignScanState *node, ParallelContext *pcxt,
                         void *coordinate);
```

Инициализирует динамическую разделяемую память, которая потребуется для параллельной операции. coordinate указывает на область разделяемой памяти размера, равного возвращаемому значению EstimateDSMForeignScan. Эта функция является необязательной и может быть опущена, если не требуется.

```
void
ReInitializeDSMForeignScan(ForeignScanState *node, ParallelContext *pcxt,
                           void *coordinate);
```

Заново инициализирует динамическую разделяемую память, требуемую для параллельной операции, перед тем как будет повторно просканирован узел чтения сторонних данных. Эта функция является необязательной и может быть опущена, если не требуется. В этой функции рекомендуется сбрасывать только общее состояние, а в функции `ReScanForeignScan` сбрасывать только локальное. В настоящее время эта функция будет вызываться перед `ReScanForeignScan`, но лучше на этот порядок не рассчитывать.

```
void
InitializeWorkerForeignScan(ForeignScanState *node, shm_toc *toc,
                            void *coordinate);
```

Инициализирует локальное состояние параллельного исполнителя на основе общего состояния, заданного ведущим исполнителем во время `InitializeDSMForeignScan`. Эта функция является необязательной и может быть опущена, если не требуется.

```
void
ShutdownForeignScan(ForeignScanState *node);
```

Освобождает ресурсы, когда становится понятно, что этот узел больше не будет выполняться. Этот обработчик вызывается не во всех случаях; иногда может вызываться только `EndForeignScan`. Так как сегмент DSM, используемый параллельным запросом, освобождается сразу после вызова этого обработчика, обёртки сторонних данных, которым нужно выполнять некоторые действия до ликвидации сегмента DSM, должны реализовывать этот метод.

52.3. Вспомогательные функции для обёрток сторонних данных

Ядро сервера экспортирует набор полезных вспомогательных функций, которые позволяют разработчикам обёрток сторонних данных легко обращаться к атрибутам объектов, связанных с FDW, например, к параметрам FDW. Чтобы использовать эти функции, необходимо включить в исходный файл заголовочный файл *foreign/foreign.h*. В этом заголовочном файле также определяются типы структур, возвращаемых этими функциями.

```
ForeignDataWrapper *
GetForeignDataWrapper(Oid fdwid);
```

Эта функция возвращает объект `ForeignDataWrapper` для обёртки сторонних данных с указанным OID. Объект `ForeignDataWrapper` содержит свойства FDW (они описаны в `foreign/foreign.h`).

```
ForeignServer *
GetForeignServer(Oid serverid);
```

Эта функция возвращает объект `ForeignServer` для стороннего сервера с указанным OID. Объект `ForeignServer` содержит свойства сервера (они описаны в `foreign/foreign.h`).

```
UserMapping *
GetUserMapping(Oid userid, Oid serverid);
```

Эта функция возвращает объект `UserMapping` для сопоставления пользователя, которое определено для указанной роли на указанном сервере. (Если сопоставление для указанной роли отсутствует, она возвращает сопоставление для `PUBLIC` или выдаёт ошибку, если его нет.) Объект `UserMapping` содержит свойства сопоставления пользователя (они описаны в `foreign/foreign.h`).

```
ForeignTable *
GetForeignTable(Oid relid);
```

Эта функция возвращает объект `ForeignTable` для сторонней таблицы с указанным OID. Объект `ForeignTable` содержит свойства сторонней таблицы (они описаны в `foreign/foreign.h`).

```
List *
GetForeignColumnOptions(Oid relid, AttrNumber attnum);
```

Эта функция возвращает параметры FDW уровня столбцов для столбца из таблицы с указанным OID сторонней таблицы и указанным номером, в виде списка `DefElem`. Если для столбца не определены параметры, возвращается NULL.

В дополнение к функциям, выбирающим объекты по OID, для некоторых объектов добавлены функции поиска по именам:

```
ForeignDataWrapper *
GetForeignDataWrapperByName(const char *name, bool missing_ok);
```

Эта функция возвращает объект `ForeignDataWrapper` для обёртки сторонних данных с указанным именем. В случае отсутствия такой обёртки возвращается NULL, если missing_ok равно true, а иначе выдаётся ошибка.

```
ForeignServer *
GetForeignServerByName(const char *name, bool missing_ok);
```

Эта функция возвращает объект `ForeignServer` для стороннего сервера с указанным именем. В случае отсутствия такого сервера возвращается NULL, если missing_ok равно true, а иначе выдаётся ошибка.

52.4. Планирование запросов с обёртками сторонних данных

Процедуры в FDW, реализующие функции `GetForeignRelSize`, `GetForeignPaths`, `GetForeignPlan`, `PlanForeignModify`, `GetForeignJoinPaths`, `GetForeignUpperPaths` и `PlanDirectModify`, должны вписываться в работу планировщика Postgres Pro. Здесь даётся несколько замечаний о том, как это должно происходить.

Для уменьшения объёма выбираемых из сторонней таблицы данных (и как следствие, сокращения стоимости) может использоваться информация, поступающая в `root` и `baserel`. Особый интерес представляет поле `baserel->baserestrictinfo`, так как оно содержит ограничивающие условия (предложение `WHERE`), по которым можно отфильтровать выбираемые строки. (Сама FDW не обязательно должна применять эти ограничения, так как их может проверить и ядро исполнителя.) Список `baserel->reltarget->exprs` позволяет определить, какие именно столбцы требуется выбрать; но учтите, что в нём перечисляются только те столбцы, которые выдаются узлом плана `ForeignScan`, но не столбцы, которые задействованы в ограничивающих условиях и при этом не выводятся запросом.

Когда функциям планирования FDW требуется сохранять свою информацию, они могут использовать различные частные поля. Вообще, все структуры, которые FDW помещает в закрытые поля, должны выделяться функцией palloc, чтобы они автоматически освобождались при завершении планирования.

Для хранения информации, относящейся к определённой сторонней таблице, функции планирования FDW могут использовать поле `baserel->fdw_private`, которое может содержать указатель на `void`. Ядро планировщика никак не касается его, кроме того, что записывает в него NULL при создании узла `RelOptInfo`. Оно полезно для передачи информации из `GetForeignRelSize` в `GetForeignPaths` и/или из `GetForeignPaths` в `GetForeignPlan` и позволяет избежать повторных вычислений.

`GetForeignPaths` может обозначить свойства различных путей доступа, сохранив частную информацию в поле `fdw_private` узлов `ForeignPath`. Это поле `fdw_private` объявлено как указатель на список (`List`), но в принципе может содержать всё, что угодно, так как ядро планировщика его не касается. Однако лучше поместить в него данные, которые сможет представить функция `nodeToString`, для применения средств отладки, имеющихся на сервере.

GetForeignPlan может изучить поле fdw_private выбранного узла ForeignPath и сформировать списки fdw_exprs и fdw_private, которые будут помещены в узел ForeignScan, где они будут находиться во время выполнения запроса. Оба эти списка должны быть представлены в форме, которую способна копировать функция copyObject. Список fdw_private не имеет других ограничений и никаким образом не интерпретируется ядром сервера. Список fdw_exprs, если этот указатель не NULL, предположительно содержит деревья выражений, которые должны быть вычислены при выполнении запроса. Затем планировщик обрабатывает эти деревья, чтобы они были полностью готовы к выполнению.

GetForeignPlan обычно может скопировать полученный целевой список в узел плана как есть. Передаваемый список scan_clauses содержит те же предложения, что и baserel->baserestrictinfo, но, возможно, в другом порядке для более эффективного выполнения. В простых случаях FDW может просто убрать узлы RestrictInfo из списка scan_clauses (используя функцию extract_actual_clauses) и поместить все предложения в список ограничений узла плана, что будет означать, что эти предложения будут проверяться исполнителем во время выполнения. Более сложные FDW могут самостоятельно проверять некоторые предложения, и в этом случае такие предложения можно удалить из списка ограничений узла, чтобы исполнитель не тратил время на их перепроверку.

Например, FDW может распознавать некоторые предложения ограничений вида *сторонняя_переменная = подвыражение*, которые, по её представлению, могут выполняться на удалённом сервере с локально вычисленным значением *подвыражения*. Собственно выявление такого предложения должно происходить в функции GetForeignPaths, так как это влияет на оценку стоимости пути. Эта функция может включить в поле fdw_private конкретного пути указатель на узел RestrictInfo этого предложения. Затем GetForeignPlan удалит это предложение из scan_clauses, но добавит *подвыражение* в fdw_exprs, чтобы оно было приведено к исполняемой форме. Она также может поместить управляющую информацию в поле fdw_private плана узла, которая скажет исполняющим функциям, что делать во время выполнения. Запрос, передаваемый удалённому серверу, будет содержать что-то вроде WHERE *сторонняя_переменная* = $1, а значение параметра будет получено во время выполнения в результате вычисления дерева выражения fdw_exprs.

Все предложения, удаляемые из списка условий узла плана, должны быть добавлены в fdw_recheck_quals или перепроверены функцией RecheckForeignScan для обеспечения корректного поведения на уровне изоляции READ COMMITTED. Когда имеет место параллельное изменение в некоторой другой таблице, задействованной в запросе, исполнителю может потребоваться убедиться в том, что все исходные условия по-прежнему выполняются для кортежа, возможно, с другим набором значений параметров. Использовать fdw_recheck_quals обычно проще, чем реализовывать проверки внутри RecheckForeignScan, но этот метод недостаточен, когда внешние соединения выносятся наружу, так как вследствие перепроверки в соединённых кортежах могут обнуляться некоторые поля, но сами кортежи не будут исключаться.

Ещё одно поле ForeignScan, которое могут заполнять FDW, это fdw_scan_tlist, описывающее кортежи, возвращаемые обёрткой для этого узла плана. Для простых сторонних таблиц в него можно записать NIL, из чего будет следовать, что возвращённые кортежи имеют тип, объявленный для сторонней таблицы. Отличное от NIL значение должно указывать на список целевых элементов (список структур TargetEntry), содержащий переменные и/или выражения, представляющие возвращаемые столбцы. Это можно использовать, например, чтобы показать, что FDW опустила некоторые столбцы, которые по её наблюдению не нужны для запроса. Также, если FDW может вычислить выражения, используемые в запросе, более эффективно, чем это можно сделать локально, она должна добавить эти выражения в список fdw_scan_tlist. Заметьте, что планы соединения (полученные из путей, созданных функцией GetForeignJoinPaths) должны всегда заполнять fdw_scan_tlist, описывая набор столбцов, которые они будут возвращать.

FDW должна всегда строить минимум один путь, зависящий только от предложений ограничения таблицы. В запросах с соединением она может также построить пути, зависящие от ограничения соединения, например *сторонняя_переменная = локальная_переменная*. Такие предложения будут отсутствовать в baserel->baserestrictinfo; их нужно искать в списках соединений

отношений. Путь, построенный с таким предложением, называется «параметризованным». Другие отношения, задействованные в выбранном предложении соединения, должны связываться с этим путём соответствующим значением `param_info`; для получения этого значения используется `get_baserel_parampathinfo`. В `GetForeignPlan` часть *`локальная_переменная`* предложения соединения будет добавлена в `fdw_exprs`, и затем, во время выполнения, это будет работать так же, как и обычное предложение ограничения.

Если FDW поддерживает удалённые соединения, `GetForeignJoinPaths` должна выдавать пути `ForeignPath` для потенциально удалённых соединений почти так же, как это делает `GetForeignPaths` для базовых таблиц. Информация о выбранном соединении может быть передана функции `GetForeignPlan` так же, как было описано выше. Однако поле `baserestrictinfo` неприменимо к отношениям соединения; вместо этого соответствующие предложения соединения для конкретного соединения передаются в `GetForeignJoinPaths` в отдельном параметре (`extra->restrictlist`).

FDW может дополнительно поддерживать прямое выполнение некоторых действий плана, находящихся выше уровня сканирований и соединений, например, группировки или агрегирования. Для реализации этой возможности FDW должна сформировать пути и вставить их в соответствующее *верхнее отношение*. Например, путь, представляющий удалённое агрегирование, должен вставляться в отношение `UPPERREL_GROUP_AGG` с помощью `add_path`. Этот путь будет сравниваться по стоимости с локальным агрегированием, выполненным по результатам пути простого сканирования стороннего отношения (заметьте, что такой путь также должен быть сформирован, иначе во время планирования произойдёт ошибка). Если путь с удалённым агрегированием выигрывает, что, как правило, и происходит, он будет преобразован в план обычным образом, вызовом `GetForeignPlan`. Такие пути рекомендуется формировать в обработчике `GetForeignUpperPaths`, который вызывается для каждого верхнего отношения (то есть на каждом шаге обработки после сканирования/соединения), если все базовые отношения запроса выдаются одной обёрткой.

`PlanForeignModify` и другие обработчики, описанные в Подразделе 52.2.4, рассчитаны на то, что стороннее отношение будет сканироваться обычным способом, а затем отдельные изменения строк будут обрабатываться локальным узлом плана `ModifyTable`. Этот подход необходим в общем случае, когда для такого изменения требуется прочитать не только сторонние, но и локальные таблицы. Однако если операция может быть целиком выполнена сторонним сервером, FDW может построить путь, представляющий эту возможность, и вставить его в верхнее отношение `UPPERREL_FINAL`, где он будет конкурировать с подходом `ModifyTable`. Этот подход также должен применяться для реализации удалённого `SELECT FOR UPDATE`, вместо обработчиков блокировки строк, описанных Подразделе 52.2.5. Учтите, что путь, вставляемый в `UPPERREL_FINAL`, отвечает за реализацию *всех* аспектов поведения запроса.

При планировании запросов `UPDATE` или `DELETE` функции `PlanForeignModify` и `PlanDirectModify` могут обратиться к структуре `RelOptInfo` сторонней таблицы и воспользоваться информацией `baserel->fdw_private`, записанной ранее функциями планирования сканирования. Однако при запросе `INSERT` целевая таблица не сканируется, так что для неё `RelOptInfo` не заполняется. На список (`List`), возвращаемый функцией `PlanForeignModify`, накладываются те же ограничения, что и на список `fdw_private` в узле плана `ForeignScan`, то есть он должен содержать только такие структуры, которые способна копировать функция `copyObject`.

Команда `INSERT` с предложением `ON CONFLICT` не поддерживает указание объекта конфликта, так как уникальные ограничения или ограничения-исключения в удалённых таблицах неизвестны локально. Из этого, в свою очередь, вытекает, что предложение `ON CONFLICT DO UPDATE` не поддерживается, так как в нём это указание является обязательным.

52.5. Блокировка строк в обёртках сторонних данных

Если нижележащий механизм хранения FDW поддерживает концепцию блокировки отдельных строк, предотвращающую одновременное изменение этих строк, обычно имеет смысл реализовать в FDW установление блокировок на уровне строк в приближении, настолько близком к обычным

таблицам Postgres Pro, насколько это возможно и практично. При этом нужно учитывать ряд замечаний.

Первое важное решение, которое нужно принять — будет ли реализована *ранняя блокировка* или *поздняя блокировка*. С ранней блокировкой строка блокируется, когда впервые считывается из нижележащего хранилища, тогда как с поздней блокировкой строка блокируется, только когда известно, что её нужно заблокировать. (Различие возникает из-за того, что некоторые строки могут быть отброшены локально проверяемыми условиями ограничений или соединений.) Ранняя блокировка гораздо проще и не требует дополнительных обращений к удалённому хранилищу, но может вызывать блокировку строк, которые можно было бы не блокировать, что может повлечь учащение конфликтов и даже неожиданные взаимоблокировки. Кроме того, поздняя блокировка возможна, только если блокируемая строка может быть однозначно идентифицирована позже. Поэтому в идентификаторе строки следует идентифицировать определённую версию строки, как это делает TID в Postgres Pro.

По умолчанию Postgres Pro игнорирует возможности блокировки, обращаясь к FDW, но FDW может установить ранние блокировки и без явной поддержки со стороны ядра. Функции, описанные в Подразделе 52.2.5, которые были добавлены в API в Postgres Pro 9.5, позволяют FDW применять поздние блокировки, если она этого пожелает.

Также следует учесть, что в режиме изоляции `READ COMMITTED` серверу Postgres Pro может потребоваться перепроверить условия ограничений и соединения с изменённой версией некоторого целевого кортежа. Для перепроверки условий соединения требуется повторно получить копии исходных строк, которые ранее были соединены в целевой кортеж. В случае со стандартными таблицами Postgres Pro для этого в список столбцов, проходящих через соединение, включаются TID из исходных таблиц, а затем исходные строки извлекаются заново при необходимости. При таком подходе набор данных соединения остаётся компактным, но требуется недорогая операция повторного чтения строк, а также возможность однозначно идентифицировать повторно считываемую версию строки по TID. Поэтому по умолчанию при работе со сторонними таблицами в список столбцов, проходящих через соединение, включается копия всей строки, извлекаемой из сторонней таблицы. Это не накладывает специальных требований на FDW, но может привести к снижению производительности при соединении слиянием или по хешу. FDW, которая может удовлетворить требованиям повторного чтения, может реализовать первый вариант.

Для команд `UPDATE` или `DELETE` со сторонней таблицей рекомендуется, чтобы операция `ForeignScan` в целевой таблице выполняла раннюю блокировку строк, которые она выбирает, возможно, используя аналог `SELECT FOR UPDATE`. FDW может определить, является ли таблица целевой таблицей команд `UPDATE`/`DELETE`, во время планирования, сравнив её relid с `root->parse->resultRelation`, или во время планирования, вызвав `ExecRelationIsTargetRelation()`. Также возможно выполнять позднюю блокировку в обработчике `ExecForeignUpdate` или `ExecForeignDelete`, но специальной поддержки для этого нет.

Для сторонних таблиц, блокировка которых запрашивается командой `SELECT FOR UPDATE/SHARE`, операция `ForeignScan` так же может произвести раннюю блокировку, выбрав кортежи, используя аналог `SELECT FOR UPDATE/SHARE`. Чтобы вместо этого произвести позднюю блокировку, предоставьте подпрограммы-обработчики, описанные в Подразделе 52.2.5. В `GetForeignRowMarkType` выберите вариант отметки строк `ROW_MARK_EXCLUSIVE`, `ROW_MARK_NOKEYEXCLUSIVE`, `ROW_MARK_SHARE` или `ROW_MARK_KEYSHARE`, в зависимости от запрошенной силы блокировки. (Код ядра будет работать одинаково при любом из этих четырёх вариантов.) Затем вы сможете определить, должна ли сторонняя таблица блокироваться командой этого типа, вызвав функцию `get_plan_rowmark` во время планирования либо `ExecFindRowMark` во время выполнения; нужно проверить не только, что возвращённая структура rowmark отлична от NULL, но и что её поле `strength` не равно `LCS_NONE`.

Наконец, для сторонних таблиц, задействованных в командах `UPDATE`, `DELETE` или `SELECT FOR UPDATE/SHARE`, но не требующих блокировки строк, можно переопределить поведение по умолчанию, заключающееся в копировании строк целиком, выбрав в `GetForeignRowMarkType`

вариант ROW_MARK_REFERENCE, получив значение силы блокировки LCS_NONE. В результате RefetchForeignRow будет вызываться с таким значением markType; она должна будет заново считывать строку, не запрашивая новую блокировку. (Если вы реализуете функцию GetForeignRowMarkType, но не хотите повторно считывать незаблокированные строки, выберите для LCS_NONE вариант ROW_MARK_COPY.)

Дополнительные сведения можно получить в src/include/nodes/lockoptions.h, в комментариях к RowMarkType и PlanRowMark в src/include/nodes/plannodes.h, и в комментариях к ExecRowMark в src/include/nodes/execnodes.h.

Глава 53. Написание метода извлечения выборки таблицы

Реализация предложения TABLESAMPLE в Postgres Pro поддерживает подключение собственных методов извлечения выборки таблицы, в дополнение к методам BERNOULLI и SYSTEM, которые требуются стандартом SQL. Метод выборки определяет, какие строки таблицы будут выбираться, когда используется предложение TABLESAMPLE.

На уровне SQL метод извлечения выборки таблицы представляется одной функцией SQL, обычно реализуемой на C, имеющей сигнатуру

```
method_name(internal) RETURNS tsm_handler
```

Имя функции будет совпадать с именем метода, указываемым в предложении TABLESAMPLE. Аргумент internal является фиктивным (в нём всегда передаётся ноль) и введён только для того, чтобы эту функцию нельзя было вызывать напрямую из команд SQL. Возвращать эта функция должна структуру типа TsmRoutine (выделенную вызовом palloc), содержащую указатели на опорные функции для метода извлечения выборки. Эти опорные функции представляют собой простые функции на C, которые не видны и не могут вызываться на уровне SQL. Эти опорные функции описаны в Разделе 53.1.

В дополнение к указателям на функции в структуре TsmRoutine должны задаваться следующие дополнительные поля:

`List *parameterTypes`

Это список OID, содержащий OID типов данных параметров, которые будут приниматься предложением TABLESAMPLE при использовании этого метода извлечения выборки. Например, для встроенных методов этот список содержит один элемент со значением FLOAT4OID, представляющий процент выборки. Другие методы могут иметь дополнительные или иные параметры.

`bool repeatable_across_queries`

Если это поле равно true, данный метод извлечения выборки может выдавать одинаковые выборки при последовательных запросах с одними и теми же параметрами и значением затравки REPEATABLE при условии неизменности содержимого таблицы. Если равно false, предложение REPEATABLE не будет приниматься с этим методом извлечения выборки.

`bool repeatable_across_scans`

Если это поле равно true, метод извлечения выборки может выдавать одинаковые выборки при последовательном сканировании в рамках одного запроса (предполагается неизменность параметров, значения затравки и снимка данных). Если равно false, планировщик не будет выбирать планы, требующие неоднократного сканирования выборки, так как это может привести к несогласованному результату запроса.

Тип структуры TsmRoutine объявлен в src/include/access/tsmapi.h, где можно найти дополнительную информацию.

Методы извлечения выборки, включённые в стандартный дистрибутив, могут послужить хорошим примером, если вы хотите написать свой метод. Код встроенных методов вы можете найти в подкаталоге src/backend/access/tablesample дерева исходного кода, а код дополнительных методов — в подкаталоге contrib.

53.1. Опорные функции метода извлечения выборки

Функция-обработчик TSM возвращает структуру TsmRoutine (выделенную вызовом palloc) с указателями на опорные функции, описанные ниже. Большинство этих функций обязательные, но некоторые — нет, и их указатели могут быть равны NULL.

```
void
SampleScanGetSampleSize (PlannerInfo *root,
                         RelOptInfo *baserel,
                         List *paramexprs,
                         BlockNumber *pages,
                         double *tuples);
```

Эта функция вызывается во время планирования. Она должна рассчитать число страниц отношения, которые будут прочитаны при простом сканировании, и число кортежей, выбираемых при сканировании. (Например, эти числа можно получить, оценив процент выбираемых данных, а затем умножив `baserel->pages` и `baserel->tuples` на это значение и округлив результат до целых.) Список `paramexprs` содержит выражения, переданные в параметрах предложению `TABLESAMPLE`. Если для целей оценивания нужны их значения, рекомендуется воспользоваться `estimate_expression_value()`, чтобы попытаться свести эти выражения к константам; но данная функция должна выдавать оценку размера, даже если это не удастся, и не должна выдавать ошибку, даже если считает переданные значения неверными (помните, что это только приблизительные оценки чисел, которые будут получены во время выполнения). Параметры `pages` и `tuples` являются выходными.

```
void
InitSampleScan (SampleScanState *node,
                int eflags);
```

Выполняет инициализацию перед выполнением узла плана SampleScan. Эта функция вызывается при запуске исполнителя. Она должна выполнить все подготовительные действия, необходимые для начала обработки. Узел `SampleScanState` уже был создан, но его поле `tsm_state` содержит NULL. Функция `InitSampleScan` может выделить через palloc область для любых внутренних данных, нужных методу извлечения выборки, и сохранить указатель на неё в `node->tsm_state`. Информацию о сканируемой таблице можно получить через другие поля узла `SampleScanState` (но заметьте, что дескриптор сканирования `node->ss.ss_currentScanDesc` ещё не настроен). Параметр `eflags` содержит битовые флаги, описывающие режим работы исполнителя для этого узла плана.

Когда `(eflags & EXEC_FLAG_EXPLAIN_ONLY)` не равно нулю, собственно сканирование не будет выполняться, поэтому эта функция должна сделать только то, что необходимо для получения состояния узла, подходящего для `EXPLAIN` и `EndSampleScan`.

Эту функцию можно опустить (присвоить указателю NULL), тогда вся инициализация, необходимая для метода извлечения выборки, должна иметь место в `BeginSampleScan`.

```
void
BeginSampleScan (SampleScanState *node,
                 Datum *params,
                 int nparams,
                 uint32 seed);
```

Начинает выполнение сканирования выборки. Эта функция вызывается непосредственно перед первой попыткой выбрать кортеж и может вызываться повторно, если потребуется перезапустить сканирование. Информацию о сканируемой таблице можно получить через поля узла `SampleScanState` (но заметьте, что дескриптор сканирования `node->ss.ss_currentScanDesc` ещё не настроен). Массив `params`, длины `nparams`, содержит значения параметров, переданных в предложении `TABLESAMPLE`. Их количество и типы задаются в списке `parameterTypes` метода выборки, и они гарантированно не равны NULL. Параметр `seed` содержит значение затравки, которое этот метод должен учитывать при генерации любых случайных чисел; это либо хеш, полученный из значения `REPEATABLE`, если оно было передано, либо результат `random()` в противном случае.

Эта функция может скорректировать поля `node->use_bulkread` и `node->use_pagemode`. Если поле `node->use_bulkread` равно `true` (это значение по умолчанию), при сканировании будет

использоваться стратегия доступа к буферу, ориентированная на переработку буферов после использования. Может быть разумным присвоить ему `false`, если при сканировании будет просматриваться только небольшой процент страниц. Если поле `node->use_pagemode` равно `true` (это значение по умолчанию), при сканировании проверка видимости будет выполняться в один проход для всех кортежей на каждой просматриваемой странице. Может иметь смысл присвоить ему `false`, если при сканировании выбирается только небольшой процент кортежей на странице. В результате будет выполняться меньше проверок видимости кортежей, хотя каждая проверка будет дороже, так как потребует расширенную блокировку.

Если метод выборки помечен как `repeatable_across_scans`, он должен быть способен выбирать при повторном сканировании тот же набор кортежей, что был выбран в первый раз, то есть новый вызов `BeginSampleScan` должен приводить к выборке тех же кортежей, что и предыдущий (если параметры `TABLESAMPLE` и значение затравки не меняются).

```
BlockNumber
NextSampleBlock (SampleScanState *node);
```

Возвращает номер блока следующей сканируемой страницы либо `InvalidBlockNumber`, если страниц для сканирования не осталось.

Эту функцию можно опустить (присвоить её указателю NULL), в этом случае код ядра произведёт последовательное сканирование всего отношения. Такое сканирование может быть синхронизированным, так что метод выборки не должен полагать, что страницы отношения каждый раз просматриваются в одном и том же порядке.

```
OffsetNumber
NextSampleTuple (SampleScanState *node,
                 BlockNumber blockno,
                 OffsetNumber maxoffset);
```

Возвращает номер смещения следующего кортежа, выбираемого с указанной страницы, либо `InvalidOffsetNumber`, если кортежей для выборки не осталось. В `maxoffset` задаётся максимальный номер смещения, допустимый на этой странице.

Примечание

`NextSampleTuple` не говорит явно, для каких из номеров смещений в диапазоне `1 .. maxoffset` действительно содержатся актуальные кортежи. Это обычно не проблема, так как код ядра игнорирует запросы на выборку несуществующих или невидимых кортежей; это не должно приводить к отклонениям в выборке. Однако при необходимости функция может проверить `node->ss.ss_currentScanDesc->rs_vistuples[]` и понять, какие кортежи актуальны и видимы. (Для этого требуется, чтобы признак `node->use_pagemode` равнялся `true`.)

Примечание

Функция `NextSampleTuple` *не* должна полагать, что в `blockno` будет получен тот же номер страницы, что был выдан при последнем вызове `NextSampleBlock`. Этот номер определённо был выдан при каком-то предыдущем вызове `NextSampleBlock`, но код ядра может вызывать `NextSampleBlock` перед тем, как собственно сканировать страницы, для поддержки упреждающего чтения. Однако можно рассчитывать на то, что как только начнётся выборка кортежей с одной данной страницы, все последующие вызовы `NextSampleTuple` будут обращаться к этой странице, пока не будет возвращено значение `InvalidOffsetNumber`.

```
void
EndSampleScan (SampleScanState *node);
```

Завершает сканирование и освобождает ресурсы. Обычно при этом не нужно освобождать память, выделенную через palloc, но все видимые извне ресурсы должны быть очищены. Эту функцию чаще всего можно опустить (присвоить её указателю NULL), если таких ресурсов нет.

Глава 54. Написание провайдера нестандартного сканирования

Postgres Pro поддерживает набор экспериментальных средств, предназначенных для того, чтобы модули расширения могли добавлять в систему новые типы сканирования. В отличие от обёртки сторонних данных, которая должна знать, как сканировать только собственные таблицы, провайдер сканирования может реализовать нестандартный вариант сканирования любого отношения в системе. Обычно к написанию провайдера нестандартного сканирования подталкивает желание реализовать какую-то оптимизацию, не поддерживаемую основной системой, например, кеширование или аппаратное ускорение некоторого рода. В этой главе рассказывается, как написать свой провайдер нестандартного сканирования.

Процесс реализации нестандартного сканирования нового типа состоит из трёх этапов. Во-первых, во время планирования необходимо построить пути доступа, представляющие сканирование с предлагаемой стратегией. Во-вторых, если один из этих путей доступа выбирается планировщиком как оптимальная стратегия сканирования определённого отношения, этот путь доступа должен быть преобразован в план. Наконец, должно быть возможно выполнить этот план, получив при этом те же результаты, что были бы получены с любым другим путём доступа, выбранным для того же отношения.

54.1. Создание нестандартных путей сканирования

Провайдер нестандартного сканирования обычно добавляет пути для базового отношения, установив следующий обработчик, который вызывается после того, как ядро построит все пути, какие сможет построить для отношения (за исключением путей Gather, которые создаются после этого вызова с тем, чтобы они могли использовать частичные пути, добавленные данным обработчиком):

```
typedef void (*set_rel_pathlist_hook_type) (PlannerInfo *root,
                                            RelOptInfo *rel,
                                            Index rti,
                                            RangeTblEntry *rte);
extern PGDLLIMPORT set_rel_pathlist_hook_type set_rel_pathlist_hook;
```

Хотя эта функция-обработчик может изучать, изменять или удалять пути, сформированные основной системой, провайдер нестандартного сканирования обычно ограничивается созданием объектов `CustomPath` и добавлением их в `rel` (с помощью `add_path`). Провайдер нестандартного сканирования отвечает за инициализацию объекта `CustomPath`, который описан так:

```
typedef struct CustomPath
{
    Path      path;
    uint32    flags;
    List     *custom_paths;
    List     *custom_private;
    const CustomPathMethods *methods;
} CustomPath;
```

Поле `path` должно инициализироваться как для любого другого пути и включать оценку числа строк, стоимость запуска и общую, а также порядок сортировки, устанавливаемый этим путём. Поле flags задаёт битовую маску, которая должна включать флаг `CUSTOMPATH_SUPPORT_BACKWARD_SCAN`, если нестандартный путь поддерживает сканирование назад, и `CUSTOMPATH_SUPPORT_MARK_RESTORE`, если он поддерживает пометку позиции и её восстановление. Обе эти возможности являются факультативными. В необязательном поле `custom_paths` задаётся список узлов `Path`, используемых данным узлом; они будут преобразованы планировщиком в узлы `Plan`. В поле `custom_private` могут быть сохранены внутренние данные нестандартного пути. Сохранять их нужно в форме, которую может принять `nodeToString`, чтобы отладочные процедуры, пытающиеся вывести нестандартный путь, работали ожидаемым образом. Поле

`methods` должно указывать на объект (обычно статически размещённый), реализующий требуемые методы нестандартного пути (на данный момент это один метод).

Провайдер нестандартного сканирования может также реализовать пути соединений. Как и для базовых отношений, такой путь должен выдавать тот же результат, какой был бы получен обычным соединением, которое он заменяет. Для этого провайдер соединения должен установить следующий обработчик, а затем внутри функции-обработчика создать пути `CustomPath` для отношения соединения.

```
typedef void (*set_join_pathlist_hook_type) (PlannerInfo *root,
                                             RelOptInfo *joinrel,
                                             RelOptInfo *outerrel,
                                             RelOptInfo *innerrel,
                                             JoinType jointype,
                                             JoinPathExtraData *extra);
extern PGDLLIMPORT set_join_pathlist_hook_type set_join_pathlist_hook;
```

Этот обработчик будет вызываться многократно для одного отношения соединения с разными сочетаниями внутренних и внешних отношений; задача обработчика — минимизировать при этом дублирующиеся операции.

54.1.1. Обработчики пути нестандартного сканирования

```
Plan *(*PlanCustomPath) (PlannerInfo *root,
                         RelOptInfo *rel,
                         CustomPath *best_path,
                         List *tlist,
                         List *clauses,
                         List *custom_plans);
```

Преобразует нестандартный путь в законченный план. Возвращаемым значением обычно будет объект `CustomScan`, который этот обработчик должен разместить в памяти и инициализировать. За подробностями обратитесь к Разделу 54.2.

54.2. Создание нестандартных планов сканирования

Нестандартное сканирование представляется в окончательном дереве плана в виде следующей структуры:

```
typedef struct CustomScan
{
    Scan      scan;
    uint32    flags;
    List     *custom_plans;
    List     *custom_exprs;
    List     *custom_private;
    List     *custom_scan_tlist;
    Bitmapset *custom_relids;
    const CustomScanMethods *methods;
} CustomScan;
```

Объект в поле `scan` должен быть инициализирован, как и для любого другого сканирования, и включать оценки стоимости, целевые списки, условия и т. д. Поле `flags` содержит битовую маску с тем же значением, что и в `CustomPath`. В поле `custom_plans` могут быть сохранены дочерние узлы `Plan`. В `custom_exprs` могут быть сохранены деревья выражений, которые будут исправляться кодом в `setrefs.c` и `subselect.c`, а в `custom_private` следует сохранить другие внутренние данные, которые будут использоваться только самим провайдером нестандартного сканирования. Поле `custom_scan_tlist` может содержать NIL при сканировании базового отношения, что будет показывать, что нестандартное сканирование возвращает кортежи, соответствующие типу строк базового отношения. В противном случае оно должно указывать на целевой список, описывающий фактические кортежи. Список `custom_scan_tlist` должен устанавливаться при соединениях и

может задаваться при сканировании, если провайдер сканирования может вычислять какие-либо выражения без переменных. Поле `custom_relids` заполняется ядром и задаёт набор отношений (индексов в списке отношений), которые обрабатывает данный узел сканирования; когда имеет место сканирование, а не соединение, в этом списке будет всего один элемент. Поле `methods` должно указывать на объект (обычно статически размещённый), реализующий требуемые методы нестандартного сканирования, которые подробнее описываются ниже.

Когда `CustomScan` сканирует одно отношение, в `scan.scanrelid` должен задаваться индекс сканируемой таблицы в списке отношений. Когда он заменяет соединение, поле `scan.scanrelid` должно быть нулевым.

Деревья планов должны поддерживать возможность копирования функцией `copyObject`, так что все данные, сохранённые в «дополнительных» полях, должны быть узлами, которые может обработать эта функция. Более того, провайдеры нестандартного сканирования не могут заменять структуру `CustomScan` расширенной структурой, её содержащей, что возможно с `CustomPath` или `CustomScanState`.

54.2.1. Обработчики плана нестандартного сканирования

```
Node *(*CreateCustomScanState) (CustomScan *cscan);
```

Выделяет структуру `CustomScanState` для заданного объекта `CustomScan`. Фактически выделенная область будет обычно больше, чем требуется для самой структуры `CustomScanState`, так как многие провайдеры могут включать её в расширенную структуру в качестве первого поля. В возвращаемом значении должны быть подходящим образом заполнены тег узла и поле `methods`, но другие поля на данном этапе должны быть обнулены; после того как `ExecInitCustomScan` произведёт базовую инициализацию, будет вызван обработчик `BeginCustomScan`, в котором провайдер нестандартного сканирования может выполнить все остальные требуемые действия.

54.3. Выполнение нестандартного сканирования

Когда выполняется узел `CustomScan`, его состояние представляется структурой `CustomScanState`, объявленной следующим образом:

```
typedef struct CustomScanState
{
    ScanState ss;
    uint32    flags;
    const CustomExecMethods *methods;
} CustomScanState;
```

Поле `ss` инициализируется как и для состояния любого другого сканирования, за исключением того, что когда это сканирование для соединения, а не для базового отношения, в `ss.ss_currentRelation` остаётся NULL. Поле `flags` содержит битовую маску с тем же значением, что и в `CustomPath` и `CustomScan`. Поле `methods` должно указывать на объект (обычно статически размещённый), реализующий требуемые методы состояния нестандартного сканирования, подробнее описанные ниже. Обычно структура `CustomScanState`, которой не нужно поддерживать `copyObject`, фактически включается в расширенную структуру в качестве её первого члена.

54.3.1. Обработчики выполнения нестандартного сканирования

```
void (*BeginCustomScan) (CustomScanState *node,
                         EState *estate,
                         int eflags);
```

Завершает инициализацию переданного объекта `CustomScanState`. Стандартные поля инициализируются в `ExecInitCustomScan`, но все внутренние поля должны инициализироваться здесь.

```
TupleTableSlot *(*ExecCustomScan) (CustomScanState *node);
```

Считывает следующий кортеж. В случае наличия кортежей эта функция должна записать в `ps_ResultTupleSlot` следующий кортеж в текущем направлении сканирования и вернуть слот с кортежем. Если же кортежей больше нет, она должна вернуть `NULL` или пустой слот.

```
void (*EndCustomScan) (CustomScanState *node);
```

Очищает все внутренние данные, связанные с `CustomScanState`. Этот метод является обязательным, но он может ничего не делать, если такие данные отсутствуют или они будут очищены автоматически.

```
void (*ReScanCustomScan) (CustomScanState *node);
```

Возвращает позицию текущего сканирования в начало и подготавливает повторное сканирование отношения.

```
void (*MarkPosCustomScan) (CustomScanState *node);
```

Сохраняет текущую позицию сканирования, чтобы к ней впоследствии можно было вернуться, вызвав обработчик `RestrPosCustomScan`. Данный обработчик является необязательным и должен присутствовать, только если установлен флаг `CUSTOMPATH_SUPPORT_MARK_RESTORE`.

```
void (*RestrPosCustomScan) (CustomScanState *node);
```

Восстанавливает предыдущую позицию сканирования, сохранённую обработчиком `MarkPosCustomScan`. Данный обработчик является необязательным и должен присутствовать, только если установлен флаг `CUSTOMPATH_SUPPORT_MARK_RESTORE`.

```
Size (*EstimateDSMCustomScan) (CustomScanState *node,
                               ParallelContext *pcxt);
```

Оценивает объём динамической разделяемой памяти, которая потребуется для параллельной операции. Это значение может превышать объём, который будет занят фактически, но не должно быть меньше. Возвращаемое значение задаётся в байтах. Этот обработчик не является обязательным и должен устанавливаться, только если провайдер нестандартного сканирования поддерживает параллельное выполнение.

```
void (*InitializeDSMCustomScan) (CustomScanState *node,
                                 ParallelContext *pcxt,
                                 void *coordinate);
```

Инициализирует динамическую разделяемую память, которая потребуется для параллельной операции; `coordinate` указывает на область разделяемой памяти размера, равного возвращаемому значению `EstimateDSMCustomScan`. Этот обработчик является необязательным и должен устанавливаться, только если провайдер нестандартного сканирования поддерживает параллельное выполнение.

```
void (*ReInitializeDSMCustomScan) (CustomScanState *node,
                                   ParallelContext *pcxt,
                                   void *coordinate);
```

Заново инициализирует динамическую разделяемую память, требуемую для параллельной операции, перед тем как будет повторно просканирован узел нестандартного сканирования. Этот обработчик является необязательным и должен устанавливаться, только если провайдер нестандартного сканирования поддерживает параллельное выполнение. В этом обработчике рекомендуется сбрасывать только общее состояние, а в обработчике `ReScanCustomScan` сбрасывать только локальное. В настоящее время этот обработчик будет вызываться перед `ReScanCustomScan`, но лучше на этот порядок не рассчитывать.

```
void (*InitializeWorkerCustomScan) (CustomScanState *node,
                                    shm_toc *toc,
                                    void *coordinate);
```

Инициализирует локальное состояние параллельного исполнителя на основе общего состояния, заданного ведущим исполнителем во время `InitializeDSMCustomScan`. Этот обработчик является

необязательным и должен устанавливаться, только если провайдер нестандартного сканирования поддерживает параллельное выполнение.

```
void (*ShutdownCustomScan) (CustomScanState *node);
```

Освобождает ресурсы, когда становится понятно, что этот узел больше не будет выполняться. Этот обработчик вызывается не во всех случаях; иногда может вызываться только `EndCustomScan`. Так как сегмент DSM, используемый параллельным запросом, освобождается сразу после вызова этого обработчика, провайдеры нестандартного сканирования, которым нужно выполнять некоторые действия до ликвидации сегмента DSM, должны реализовывать этот метод.

```
void (*ExplainCustomScan) (CustomScanState *node,
                           List *ancestors,
                           ExplainState *es);
```

Выводит дополнительную информацию для `EXPLAIN` об узле нестандартного сканирования. Этот обработчик является необязательным. Общие данные, сохранённые в `ScanState`, такие как целевой список и сканируемое отношение, будут выводиться и без этого обработчика, но с помощью этого обработчика можно выдать дополнительные, внутренние сведения.

Глава 55. Генетический оптимизатор запросов

Автор

Разработал Мартин Утеш (<utesch@aut.tu-freiberg.de>) для Института автоматического управления в Техническом университете Фрайбергская горная академия, Германия.

55.1. Обработка запроса как сложная задача оптимизации

Среди всех реляционных операторов самым сложным для обработки и оптимизации является *соединение*. В первую очередь потому, что по мере увеличения числа соединений в запросе число возможных планов запроса увеличивается экспоненциально. Дополнительная сложность оптимизации связана с наличием различных *методов соединения* (например, в Postgres Pro это вложенный цикл, соединение по хешу и соединение слиянием) для каждого отдельного соединения и разнообразием *индексов* (например, в Postgres Pro это B-дерево, хеш, GiST и GIN), определяющих путь доступа к отношениям.

Традиционный оптимизатор запросов Postgres Pro выполняет *почти исчерпывающий поиск* во всём множестве возможных стратегий. Этот алгоритм, появившийся в СУБД IBM System R, находит порядок соединений, близкий к оптимальному, но может требовать огромного количества времени и памяти, когда число соединений оказывается большим. В результате обычный оптимизатор Postgres Pro оказывается неподходящим для запросов, в которых соединяется большое количество таблиц.

Институт автоматического управления в Техническом университете Фрайбергская горная академия, Германия, столкнулся с этими проблемами, разрабатывая систему принятия решений на основе базы знаний для обслуживания электростанций, в которой в качестве СУБД планировалось применять Postgres Pro. Для машины, делающей выводы на основе базы знаний, СУБД должна была выполнять запросы с таким количеством соединений, что использование обычного оптимизатора запросов оказалось неприемлемым.

Далее мы опишем реализацию *генетического алгоритма*, который решает проблему выбора порядка соединений эффективным способом для запросов с большим числом соединений.

55.2. Генетические алгоритмы

Генетический алгоритм (ГА) реализует метод эвристической оптимизации, построенный на случайном поиске. В данном контексте множество возможных решений проблемы оптимизации называется *популяцией особей*. Степень адаптации особи к среде определяет функция *приспособленности*.

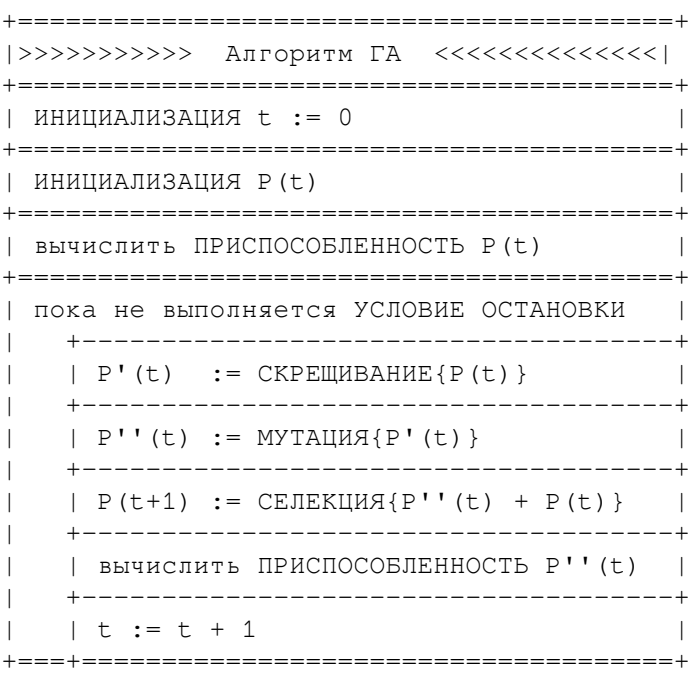
Координаты особи в пространстве поиска представляются *хромосомами*, которые по сути являются символьными строками. Фрагмент хромосомы, кодирующий значение одного оптимизируемого параметра, называется *геном*. Обычно ген кодируется в виде *двоичного* или *целочисленного* значения.

В результате симуляции эволюционных операций (*скрещивания*, *мутации* и *селекции*) данный алгоритм формирует новые поколения особей, у которых приспособленность в среднем будет выше, чем у их предшественников.

Как сказано в ответах на вопросы в группе comp.ai.genetic, нельзя не отметить, что ГА реализует не чисто случайный поиск решения проблемы. В ГА происходят вероятностные процессы, но результат явно оказывается не случайным (лучше случайного).

Рисунок 55.1. Диаграмма структуры генетического алгоритма

P(t)	поколение предков на момент t
P''(t)	поколение потомков на момент t



55.3. Генетическая оптимизация запросов (GEQO) в Postgres Pro

Модуль GEQO (Genetic Query Optimization, Генетическая оптимизация запросов) подходит к проблеме оптимизации запроса как к хорошо известной задаче коммивояжёра (TSP, Traveling Salesman Problem). Возможные планы запроса кодируются числами в строковом виде. Каждая строка представляет порядок соединения одного отношения из запроса со следующим. Например, дерево соединения

```
   /\
  /\ 2
 /\ 3
4  1
```

кодируется строкой целых чисел '4-1-3-2', которая означает: сначала соединить отношения '4' и '1', потом добавить '3', а затем '2', где 1, 2, 3, 4 — идентификаторы отношений внутри оптимизатора Postgres Pro.

Реализация GEQO в Postgres Pro имеет следующие особые характеристики:

- Использование ГА с *зафиксированным состоянием* (когда заменяются наименее приспособленные особи популяции, а не всё поколение) способствует быстрой сходимости к улучшенным планам запроса. Это важно для обработки запроса за приемлемое время;
- Использование *скрещивания с обменом рёбер*, которое очень удачно минимизирует число потерянных рёбер при решении задачи коммивояжёра с применением ГА;
- Мутация как генетический оператор считается устаревшей, так что для получения допустимых путей TSP не требуются механизмы исправления.

Части модуля GEQO взяты из алгоритма Genitor, разработанного Д. Уитли.

В результате, модуль GEQO позволяет оптимизатору запросов Postgres Pro эффективно выполнять запросы со множеством соединений, обходясь без полного перебора вариантов.

55.3.1. Построение возможных планов с GEQO

В процедуре планирования в GEQO используется код стандартного планировщика, который строит планы сканирования отдельных отношений. Затем вырабатываются планы соединений с применением генетического подхода. Как было сказано выше, каждый план соединения представляется последовательностью чисел, определяющей порядок соединений базовых отношений. На начальной стадии код GEQO просто случайным образом генерирует несколько возможных последовательностей. Затем для каждой рассматриваемой последовательности вызывается функция стандартного планировщика, оценивающая стоимость запроса в случае выбора этого порядка соединений. (Для каждого шага последовательности рассматриваются все три возможные стратегии соединения и все изначально выбранные планы сканирования отношений. Результирующей оценкой стоимости будет минимальная из всех возможных.) Последовательности соединений с наименьшей оценкой стоимости считаются «более приспособленными», чем последовательности с большей оценкой. Проанализировав возможные последовательности, генетический алгоритм отбрасывает наименее приспособленные из них. Затем генерируются новые кандидаты путём объединения генов более приспособленных последовательностей — для этого выбираются случайные фрагменты известных последовательностей с низкой стоимостью, из которых складываются новые последовательности для рассмотрения. Этот процесс повторяется, пока не будет рассмотрено некоторое предопределённое количество последовательностей соединений; после этого для построения окончательного плана выбирается лучшая последовательность, найденная за всё время поиска.

Этот процесс по природе своей недетерминирован, вследствие случайного выбора при формировании начальной популяции и последующей «мутации» лучших кандидатов. Но во избежание неожиданных изменений выбранного плана, на каждом проходе алгоритм GEQO перезапускает свой генератор случайных чисел с текущим значением параметра geqo_seed. Поэтому пока значение `geqo_seed` и другие параметры GEQO остаются неизменными, для определённого запроса (и других входных данных планировщика, в частности, статистики) будет строиться один и тот же план. Если вы хотите поэкспериментировать с разными путями соединений, попробуйте изменить `geqo_seed`.

55.3.2. Будущее развитие модуля Postgres Pro GEQO

Требуется провести дополнительную работу для выбора оптимальных параметров генетического алгоритма. В файле `src/backend/optimizer/geqo/geqo_main.c`, подпрограммах `gimme_pool_size` и `gimme_number_generations`, мы должны найти компромиссные значения параметров, удовлетворяющие двум несовместимым требованиям:

- Оптимальность плана запроса
- Время вычисления

В текущей реализации приспособленность каждой рассматриваемой последовательности соединений рассчитывается стандартным планировщиком, который каждый раз вычисляет избирательность соединения и стоимость заново. С учётом того, что различные кандидаты могут содержать общие подпоследовательности соединений, при этом будет повторяться большой объём работы. Таким образом, расчёт можно значительно ускорить, сохраняя оценки стоимости для внутренних соединений, но сложность состоит в том, чтобы уместить это состояние в разумные объёмы памяти.

На более общем уровне не вполне понятно, насколько уместно для оптимизации запросов использовать ГА, предназначенный для решения задачи коммивояжёра. В этой задаче стоимость, связанная с любой подстрокой (частью тура) не зависит от остального маршрута, но это определённо не так для оптимизации запросов. Таким образом, возникает вопрос, насколько эффективно скрещивание путём обмена рёбрами.

55.4. Дополнительные источники информации

Дополнительную информацию о генетических алгоритмах можно получить в следующих источниках:

- *The Hitch-Hiker's Guide to Evolutionary Computation*, (Руководство для путешествующих автостопом по эволюционным вычислениям, Ответы на часто задаваемые вопросы в группе *news://comp.ai.genetic*)
- *Evolutionary Computation and its application to art and design* (Эволюционные вычисления и их применение в искусстве и дизайне), Крейг Рейнольдс
- elma04
- fong

Глава 56. Определение интерфейса для индексных методов доступа

В этой главе описывается интерфейс между ядром системы Postgres Pro и *индексными методами доступа*, которые управляют отдельными типами индексов. Ядро системы не знает об индексах ничего, кроме того, что описано здесь; благодаря этому можно реализовывать абсолютно новые типы индексов в рамках расширений.

Все индексы Postgres Pro являются, говоря на техническом уровне, *вторичными индексами*; то есть, они физически отделены от файла таблицы, к которой относятся. Каждый индекс хранится в собственном отдельном физическом *отношении* и описывается в отдельной записи в каталоге `pg_class`. Содержимое индекса находится полностью под контролем соответствующего метода доступа. На практике все индексные методы доступа делят индексы на страницы стандартного размера, чтобы для обращения к содержимому индекса можно было задействовать обычный менеджер хранилища и менеджер буферов. (Более того, большинство существующих методов доступа используют одну структуру страницы, описанную в Разделе 63.6, и одинаковый формат заголовков кортежей индекса; но эти решения методам доступа не навязываются.)

Индекс по сути представляет собой сопоставление некоторых значений ключей данных с *идентификаторами кортежей*, TID (Tuple Identifier), или версиями строк в основной таблице индекса. TID состоит из номера блока и номера записи в этом блоке (см. Раздел 63.6). Этой информации достаточно, чтобы выбрать определённую версию строки из таблицы. Индексы сами по себе не знают, что в модели MVCC у одной логической строки может быть несколько существующих версий; для индекса каждый кортеж — независимый объект, которому нужна своя запись в индексе. Таким образом, при изменении строки для неё всегда заново создаются новые записи индекса, даже если значения ключа не изменились. (Кортежи HOT представляют собой исключение из этого утверждения; но индексы всё равно не имеют с этим дела.) Записи индексов для мёртвых кортежей высвобождаются (при очистке), когда высвобождаются сами мёртвые кортежи.

56.1. Базовая структура API для индексов

Каждый индексный метод доступа описывается строкой в системном каталоге `pg_am`. В записи `pg_am` указывается имя и *функция-обработчик* для метода доступа. Эти записи могут создаваться и удаляться командами SQL CREATE ACCESS METHOD и DROP ACCESS METHOD.

Функция-обработчик индексного метода доступа должна объявляться как принимающая один аргумент типа `internal` и возвращающая псевдотип `index_am_handler`. Аргумент в данном случае фиктивный, и нужен только для того, чтобы эту функцию нельзя было вызывать непосредственно из команд SQL. Возвращать эта функция должна структуру типа `IndexAmRoutine` (в памяти palloc), содержащую всё, что нужно знать коду ядра, чтобы использовать этот метод доступа. Структура `IndexAmRoutine`, также называемая *структурой API* метода доступа, содержит поля, задающие разнообразные предопределённые свойства метода доступа, например, поддерживает ли он составные индексы. Что более важно, она содержит указатели на опорные функции для метода доступа. Это обычные функции на C и они не видны и не могут быть вызваны на уровне SQL. Опорные функции описаны в Разделе 56.2.

Структура `IndexAmRoutine` определяется так:

```
typedef struct IndexAmRoutine
{
    NodeTag     type;

    /*
     * Общее число стратегий (операторов), с которыми возможен поиск/применение
     * этого метода доступа (МД).  Ноль, если у этого МД нет фиксированного набора
     * назначенных стратегий.
     */
```

```
    uint16      amstrategies;
    /* общее число опорных функций, используемых этим МД */
    uint16      amsupport;
    /* поддерживает ли МД упорядочивание (ORDER BY) значений индексированного столбца?
 */
    bool        amcanorder;
    /* поддерживает ли МД упорядочивание (ORDER BY) результата оператора с
 индексированным столбцом? */
    bool        amcanorderbyop;
    /* поддерживает ли МД сканирование в обратном направлении? */
    bool        amcanbackward;
    /* поддерживает ли МД уникальные индексы (UNIQUE)? */
    bool        amcanunique;
    /* поддерживает ли МД индексы с несколькими столбцами? */
    bool        amcanmulticol;
    /* требуется ли для сканирования с МД ограничение первого столбца индекса? */
    bool        amoptionalkey;
    /* воспринимает ли МД условия ScalarArrayOpExpr? */
    bool        amsearcharray;
    /* воспринимает ли МД условия IS NULL/IS NOT NULL? */
    bool        amsearchnulls;
    /* может ли тип, хранящийся в индексе, отличаться от типа столбца? */
    bool        amstorage;
    /* возможна ли кластеризация по индексу этого типа? */
    bool        amclusterable;
    /* МД обрабатывает предикатные блокировки? */
    bool        ampredlocks;
    /* поддерживает ли МД параллельное сканирование? */
    bool        amcanparallel;
    /* тип данных, хранящихся в индексе, либо InvalidOid, если он переменный */
    Oid         amkeytype;
    /* поддерживает ли МД включённые столбцы, задаваемые в INCLUDE? */
    bool        amcaninclude;

    /* интерфейсные функции */
    ambuild_function ambuild;
    ambuildempty_function ambuildempty;
    aminsert_function aminsert;
    ambulkdelete_function ambulkdelete;
    amvacuumcleanup_function amvacuumcleanup;
    amcanreturn_function amcanreturn;   /* может быть NULL */
    amcostestimate_function amcostestimate;
    amoptions_function amoptions;
    amproperty_function amproperty;     /* может быть NULL */
    amvalidate_function amvalidate;
    ambeginscan_function ambeginscan;
    amrescan_function amrescan;
    amgettuple_function amgettuple;     /* может быть NULL */
    amgetbitmap_function amgetbitmap;   /* может быть NULL */
    amendscan_function amendscan;
    ammarkpos_function ammarkpos;       /* может быть NULL */
    amrestrpos_function amrestrpos;     /* может быть NULL */

    /* интерфейсные функции для поддержки параллельного сканирования по индексу */
    amestimateparallelscan_function amestimateparallelscan;  /* может быть NULL */
    aminitparallelscan_function aminitparallelscan;          /* может быть NULL */
    amparallelrescan_function amparallelrescan;              /* может быть NULL */
} IndexAmRoutine;
```

Чтобы индексный метод доступа применялся, необходимо также определить *семейства операторов* и *классы операторов* в pg_opfamily, pg_opclass, pg_amop и pg_amproc. Эти записи позволяют планировщику понять, для каких видов условий запросов могут применяться индексы с данными методом доступа. Семейства и классы операторов описываются в Разделе 35.14; этот материал необходимо изучить, прежде чем читать данную главу.

Отдельный индекс определяется записью в pg_class, описывающей его как физическое отношение, и записью в pg_index, представляющей логическое содержание индекса — то есть, набор столбцов индекса и семантическое значение этих столбцов, установленное соответствующими классами операторов. Столбцами индекса (значениями ключа) могут быть либо простые столбцы нижележащей таблицы, либо выражения, вычисляемые по строкам таблицы. Для индексного метода доступа обычно не важно, откуда поступают значения ключа индекса (они всегда поступают в вычисленном виде), но очень важна информация о классе операторов в каталоге pg_index. Обе эти записи каталогов представлены в составе структуры данных Relation, которая передаётся всем функциям, реализующим операции с индексом.

С некоторыми полями флагов в IndexAmRoutine связаны неочевидные следствия. Требования индексов с amcanunique описаны в Разделе 56.5. Флаг amcanmulticol показывает, что метод доступа поддерживает составные индексы, а amoptionalkey обозначает, что метод позволяет выполнить сканирование при отсутствии индексируемого ограничивающего условия для первого столбца индекса. Когда amcanmulticol равен false, amoptionalkey по сути говорит, поддерживает ли метод доступа полное сканирование по индексу без ограничивающего условия. Методы доступа, поддерживающие индексы по нескольким столбцам, *должны* поддерживать сканирования при отсутствии ограничений любых или всех столбцов после первого; однако они могут требовать присутствия какого-либо ограничения для первого столбца индекса, и это требование отмечается значением false флага amoptionalkey. В amoptionalkey для метода доступа может устанавливаться false, например, когда этот метод доступа не индексирует значения. Так как большинство индексируемых операторов — строгие, и поэтому не могут вернуть true для операндов NULL, на первый взгляд кажется заманчивой идея не хранить записи индекса для значений NULL: они всё равно никак не могут быть прочитаны при сканировании индекса. Однако этот аргумент отпадает, когда при сканировании индекса вовсе отсутствует ограничение данного столбца индекса. На практике это означает, что индексы с установленным флагом amoptionalkey должны индексировать значения NULL, так как планировщик может склониться к использованию этого индекса вообще без ключей. С этим связано ещё одно ограничение — индексный метод доступа, поддерживающий составные индексы, *должен* поддерживать индексирование значений NULL в столбцах после первого, так как планировщик будет полагать, что индекс можно применять для запросов, в которых эти столбцы не ограничиваются. Например, рассмотрим индекс по (a,b) и запрос с ограничением WHERE a = 4. Система будет полагать, что по этому индексу можно просканировать строки с a = 4, но это будет неверно, если индекс исключит строки, в которых b — NULL. Однако этот индекс вполне может исключить строки, в которых первый столбец содержит NULL. Метод индекса, который индексирует значения NULL, может также установить флаг amsearchnulls, отметив тем самым, что он поддерживает в качестве условий поиска IS NULL и IS NOT NULL.

56.2. Функции для индексных методов доступа

Индексный метод доступа должен определить в IndexAmRoutine следующие функции построения и обслуживания индексов:

```
IndexBuildResult *
ambuild (Relation heapRelation,
         Relation indexRelation,
         IndexInfo *indexInfo);
```

Строит новый индекс. Отношение индекса уже физически создано, но пока пусто. Оно должно быть наполнено фиксированными данными, которые требуются методу доступа, и записями для всех кортежей, уже существующих в таблице. Обычно функция ambuild вызывает IndexBuildHeapScan() для поиска в таблице существующих кортежей и для вычисления ключей,

которые должны вставляться в этот индекс. Эта функция должна возвращать структуру, выделенную вызовом palloc и содержащую статистику нового индекса.

```
void
ambuildempty (Relation indexRelation);
```

Создаёт пустой индекс и записывает его в слой инициализации (INIT_FORKNUM) данного отношения. Этот метод вызывается только для нежурналируемых индексов; пустой индекс, записанный в слой инициализации, будет копироваться в основной слой отношения при каждом перезапуске сервера.

```
bool
aminsert (Relation indexRelation,
          Datum *values,
          bool *isnull,
          ItemPointer heap_tid,
          Relation heapRelation,
          IndexUniqueCheck checkUnique,
          IndexInfo *indexInfo);
```

Вставляет новый кортеж в существующий индекс. В массивах values и isnull передаются значения ключа, которые должны быть проиндексированы, а в heap_tid — идентификатор индексируемого кортежа (TID). Если метод доступа поддерживает уникальные индексы (флаг amcanunique установлен), параметр checkUnique указывает, какая проверка уникальности должна выполняться. Это зависит от того, является ли ограничение уникальности откладываемым; за подробностями обратитесь к Разделу 56.5. Обычно параметр heapRelation нужен методу доступа только для проверки уникальности (так как он должен обратиться к основным данным, чтобы убедиться в актуальности кортежа).

Возвращаемый функцией булев результат имеет значение, только когда параметр checkUnique равен UNIQUE_CHECK_PARTIAL. В этом случае результат TRUE означает, что новая запись признана уникальной, тогда как FALSE означает, что она может быть неуникальной (и требуется назначить отложенную проверку уникальности). В других случаях рекомендуется возвращать постоянный результат FALSE.

Некоторые индексы могут индексировать не все кортежи. Если кортеж не будет индексирован, aminsert должна просто завершиться, не делая ничего.

Если индексный МД хочет кешировать данные между операциями добавления в индекс в одном операторе SQL, он может выделить память в indexInfo->ii_Context и сохранить указатель на эти данные в поле indexInfo->ii_AmCache (которое изначально равно NULL).

```
IndexBulkDeleteResult *
ambulkdelete (IndexVacuumInfo *info,
              IndexBulkDeleteResult *stats,
              IndexBulkDeleteCallback callback,
              void *callback_state);
```

Удаляет кортеж(и) из индекса. Это операция «массового удаления», которая предположительно будет реализована путём сканирования всего индекса и проверки для каждой записи, должна ли она удаляться. Переданная функция callback должна вызываться в стиле callback(*TID*, callback_state) с результатом bool, который говорит, должна ли удаляться запись индекса, на которую указывает передаваемый TID. Возвращать эта функция должна NULL или структуру, выделенную вызовом palloc и содержащую статистику результата удаления. NULL можно вернуть, если никакая информация не должна передаваться в amvacuumcleanup.

Из-за ограничения maintenance_work_mem процедура ambulkdelete может вызываться несколько раз, когда удалению подлежит большое количество кортежей. В аргументе stats передаётся результат предыдущего вызова для данного индекса (при первом вызове в ходе операции VACUUM он содержит NULL). Это позволяет методу доступа накапливать статистику в процессе всей операции. Обычно ambulkdelete модифицирует и возвращает одну и ту же структуру, если в stats передаётся не NULL.

```
IndexBulkDeleteResult *
amvacuumcleanup (IndexVacuumInfo *info,
                 IndexBulkDeleteResult *stats);
```

Провести уборку после операции VACUUM (до этого ambulkdelete могла вызываться несколько или ноль раз). От этой функции не требуется ничего, кроме как выдать статистику по индексу, но она может произвести массовую уборку, например, высвободить пустые страницы индекса. В stats ей передаётся структура, возвращённая при последнем вызове ambulkdelete, либо NULL, если ambulkdelete не вызывалась, так как никакие кортежи удалять не требовалось. Эта функция должна возвращать NULL или структуру, выделенную вызовом palloc. Содержащаяся в этой структуре статистика будет отражена в записи в pg_class и попадёт в вывод команды VACUUM, если она выполнялась с указанием VERBOSE. NULL может возвращаться, если индекс вовсе не изменился в процессе операции VACUUM, но в противном случае должна возвращаться корректная статистика.

Начиная с PostgreSQL версии 8.4, amvacuumcleanup также вызывается в конце операции ANALYZE. В этом случае stats всегда NULL и любое возвращаемое значение игнорируется. Этот вариант вызова можно распознать, проверив поле info->analyze_only. При таком вызове методу доступа рекомендуется ничего не делать, кроме как провести уборку после добавления данных, и только в рабочем процессе автоочистки.

```
bool
amcanreturn (Relation indexRelation, int attno);
```

Проверяет, поддерживается ли *сканирование только индекса* для заданного столбца, когда для записи индекса возвращаются значения индексируемых столбцов в виде IndexTuple. Атрибуты нумеруются с 1, то есть для первого столбца attno равен 1. Возвращает true, если такое сканирование поддерживается, а иначе — false. Если индексный метод доступа в принципе не поддерживает сканирование только индекса, в поле amcanreturn его структуры IndexAmRoutine можно записать NULL.

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation,
                double *indexPages);
```

Рассчитывает примерную стоимость сканирования индекса. Эта функция полностью описывается ниже в Разделе 56.6.

```
bytea *
amoptions (ArrayType *reloptions,
           bool validate);
```

Разбирает и проверяет массив параметров для индекса. Эта функция вызывается, только когда для индекса задан отличный от NULL массив reloptions. Массив *reloptions* состоит из элементов типа text, содержащих записи вида *имя=значение*. Данная функция должна получить значение типа bytea, которое будет скопировано в поле rd_options записи индекса в relcache. Содержимое этого значения bytea определяется самим методом доступа; большинство стандартных методов доступа помещают в него структуру StdRdOptions. Когда параметр *validate* равен true, эта функция должна выдать подходящее сообщение об ошибке, если какие-либо параметры нераспознаны или имеют недопустимые значения; если же *validate* равен false, некорректные записи должны просто игнорироваться. (В *validate* передаётся false, когда параметры уже загружены в pg_catalog; при этом неверная запись может быть обнаружена, только если в методе доступа поменялись правила обработки параметров, и в этом случае стоит просто игнорировать такие записи.) NULL можно вернуть, когда нужно получить поведение по умолчанию.

```
bool
```

```
amproperty (Oid index_oid, int attno,
            IndexAMProperty prop, const char *propname,
            bool *res, bool *isnull);
```

Процедура `amproperty` позволяет индексным методам доступа переопределять стандартное поведение функции `pg_index_column_has_property` и связанных с ней. Если метод доступа не проявляет никаких особенностей при запросе свойств индексов, поле `amproperty` в структуре `IndexAmRoutine` может содержать NULL. В противном случае процедура `amproperty` будет вызываться с нулевыми параметрами *`index_oid`* и *`attno`* при вызове `pg_indexam_has_property`, либо с корректным *`index_oid`* и нулевым *`attno`* при вызове `pg_index_has_property`, либо с корректным *`index_oid`* и положительным *`attno`* при вызове `pg_index_column_has_property`. В *`prop`* передаётся значение перечисления, указывающее на проверяемое значение, а в *`propname`* — строка с именем свойства. Если код ядра не распознаёт имя свойства, в *`prop`* передаётся `AMPROP_UNKNOWN`. Методы доступа могут воспринимать нестандартные имена свойств, проверяя *`propname`* на совпадение (для согласованности с кодом ядра используйте для проверки `pg_strcasecmp`); для имён, известных коду ядра, лучше проверять *`prop`*. Если процедура `amproperty` возвращает `true`, это значит, что она установила результат проверки свойства: она должна задать в `*res` возвращаемое логическое значение или установить в `*isnull` значение `true`, чтобы возвратить NULL. (Перед вызовом обе упомянутые переменные инициализируются значением `false`.) Если `amproperty` возвращает `false`, код ядра переключается на обычную логику определения результата проверки свойства.

Методы доступа, поддерживающие операторы упорядочивания, должны реализовывать проверку свойства `AMPROP_DISTANCE_ORDERABLE`, так как код ядра не знает, как это сделать и возвращает NULL. Также может быть полезно реализовать проверку `AMPROP_RETURNABLE`, если это можно сделать проще, чем обращаясь к индексу и вызывая `amcanreturn` (что делает код ядра по умолчанию). Для всех остальных стандартных свойств поведение ядра по умолчанию можно считать удовлетворительным.

```
bool
amvalidate (Oid opclassoid);
```

Проверяет записи в каталоге для заданного класса операторов, насколько это может сделать метод доступа. Например, это может включать проверку, все ли необходимые опорные функции реализованы. Функция `amvalidate` должна вернуть false, если класс операторов непригоден к использованию. Сообщения о проблеме следует выдать через `ereport`.

Цель индекса, конечно, в том, чтобы поддерживать поиск кортежей, соответствующих индексируемому условию `WHERE`, по *ограничению* или *ключу поиска*. Сканирование индекса описывается более полно ниже, в Разделе 56.3. Индексный метод доступа может поддерживать «простое» сканирование, сканирование по «битовой карте» или и то, и другое. Метод доступа должен или может реализовывать следующие функции, связанные со сканированием:

```
IndexScanDesc
ambeginscan (Relation indexRelation,
             int nkeys,
             int norderbys);
```

Подготавливает метод к сканированию индекса. В параметрах `nkeys` и `norderbys` задаётся количество операторов условия и сортировки, которые будут задействованы при сканировании; это может быть полезно для выделения памяти. Заметьте, что фактические значения ключей сканирования в этот момент ещё не предоставляются. В результате функция должна выдать структуру, выделенную средствами palloc. В связи с особенностями реализации, метод доступа *должен* создать эту структуру, вызвав `RelationGetIndexScan()`. В большинстве случаев все действия `ambeginscan` сводятся только к выполнению этого вызова и, возможно, получению блокировок; всё самое интересное при запуске сканирования индекса происходит в `amrescan`.

```
void
amrescan (IndexScanDesc scan,
```

```
            ScanKey keys,
            int nkeys,
            ScanKey orderbys,
            int norderbys);
```

Запускает или перезапускает сканирование индекса, возможно, с новыми ключами сканирования. (Для перезапуска сканирования с ранее переданными ключами в `keys` и/или `orderbys` передаётся NULL.) Заметьте, что количество ключей или операторов сортировки не может превышать значения, поступившие в `ambeginscan`. На практике возможность перезапуска используется, когда в соединении со вложенным циклом выбирается новый внешний кортеж, так что требуется сравнение с новым ключом, но структура ключей сканирования не меняется.

```
bool
amgettuple (IndexScanDesc scan,
            ScanDirection direction);
```

Выбирает следующий кортеж в ходе данного сканирования, с передвижением по индексу в заданном направлении (вперёд или назад). Возвращает TRUE, если кортеж был получен, или FALSE, если подходящих кортежей не осталось. В случае успеха в структуре `scan` сохраняется TID кортежа. Заметьте, что под «успехом» здесь подразумевается только, что индекс содержит запись, соответствующую ключам сканирования, а не то, что данный кортеж обязательно существует в данных или оказывается видимым в снимке вызывающего субъекта. При положительном результате `amgettuple` должна также установить для свойства `scan->xs_recheck` значение TRUE или FALSE. FALSE будет означать, что запись индекса точно соответствует ключам сканирования, а TRUE, что есть сомнение в этом, так что условия, представленные ключами сканирования, необходимо ещё раз перепроверить для фактического кортежа, когда он будет получен. Это свойство введено для поддержки «неточных» операторов индексов. Заметьте, что такая перепроверка касается только условий сканирования; предикат частичного индекса (если он имеется) никогда не перепроверяется кодом, вызывающим `amgettuple`.

Если индекс поддерживает сканирование только индекса (то есть, `amcanreturn` для него равен TRUE), то в случае успеха метод доступа должен также проверить флаг `scan->xs_want_itup` и, если он установлен, должен вернуть исходные индексированные данные для этой записи индекса. Данные могут возвращаться посредством указателя на `IndexTuple`, сохранённого в `scan->xs_itup`, с дескриптором `scan->xs_itupdesc`; либо посредством указателя на `HeapTuple`, сохранённого в `scan->xs_hitup`, с дескриптором кортежа `scan->xs_hitupdesc`. (Второй вариант должен использоваться для восстановления данных, которые могут не уместиться в `IndexTuple`.) В любом случае за управление целевой областью данных, определяемой этим указателем, отвечает метод доступа. Данные должны оставаться актуальными как минимум до следующего вызова `amgettuple`, `amrescan` или `amendscan` в процессе сканирования.

Функция `amgettuple` должна быть реализована, только если метод доступа поддерживает «простое» сканирование индекса. В противном случае поле `amgettuple` в структуре `IndexAmRoutine` должно содержать NULL.

```
int64
amgetbitmap (IndexScanDesc scan,
             TIDBitmap *tbm);
```

Выбирает все кортежи для данного сканирования и добавляет их в передаваемую вызывающим кодом структуру `TIDBitmap` (то есть, получает логическое объединение множества TID выбранных кортежей с множеством, уже записанным в битовой карте). Возвращает эта функция число полученных кортежей (это может быть только приблизительная оценка; например, некоторые методы доступа не учитывают повторяющиеся значения). Добавляя идентификаторы кортежей в битовую карту, `amgetbitmap` может обозначить, что для этих кортежей нужно перепроверить условия сканирования. Для этого так же, как и в `amgettuple`, устанавливается выходной параметр `xs_recheck`. Замечание: в текущей реализации эта возможность увязывается с возможностью неточного хранения самих битовых карт, таким образом вызывающий код перепроверяет для отмеченных кортежей и условия сканирования, и предикат частичного индекса (если он имеется). Однако так может быть не всегда. Функции `amgetbitmap` и `amgettuple` не могут использоваться в

одном сканировании индекса; есть и другие ограничения в применении amgetbitmap, описанные в Разделе 56.3.

Функция amgetbitmap должна быть реализована, только если метод доступа поддерживает сканирование индекса «по битовой карте». В противном случае поле amgetbitmap в структуре IndexAmRoutine должно содержать NULL.

```
void
amendscan (IndexScanDesc scan);
```

Завершает сканирование и освобождает ресурсы. Саму структуру scan освобождать не следует, но любые блокировки или закрепления объектов, установленные внутри метода доступа, должны быть сняты.

```
void
ammarkpos (IndexScanDesc scan);
```

Помечает текущую позицию сканирования. Метод доступа должен поддерживать сохранение только одной позиции в процессе сканирования.

Функция ammarkpos должна быть реализована, только если метод доступа поддерживает сканирование по порядку. Если это не так, в поле ammarkpos в структуре IndexAmRoutine можно записать NULL.

```
void
amrestrpos (IndexScanDesc scan);
```

Восстанавливает позицию сканирования, отмеченную последней.

Функция amrestrpos должна быть реализована, только если метод доступа поддерживает сканирование по порядку. Если это не так, в поле amrestrpos в структуре IndexAmRoutine можно записать NULL.

Помимо обычного сканирования некоторые типы индексов могут поддерживать *параллельное сканирование индекса*, что позволяет осуществлять совместное сканирование индекса нескольким обслуживающим процессам. Для этого метод доступа должен организовать работу так, чтобы каждый из взаимодействующих процессов возвращал подмножество кортежей, которое бы возвращалось при обычном, не параллельном сканировании, и таким образом, чтобы объединение этих подмножеств совпадало с множеством кортежей, возвращаемых при обычном сканировании. Более того, чтобы не требовалась глобальная сортировка кортежей, возвращаемых при параллельном сканировании, порядок кортежей в подмножествах, выдаваемых всеми взаимодействующими процессами, должен соответствовать запрошенному. Для поддержки параллельного сканирования по индексу должны быть реализованы следующие функции:

```
Size
amestimateparallelscan (void);
```

Рассчитывает и возвращает объём (в байтах) в динамической разделяемой памяти, который может потребоваться для осуществления параллельного сканирования. (Этот объём дополняет, а не заменяет объём памяти, затребованный для данных, независимо от МД, в ParallelIndexScanDescData.)

Эту функцию можно не реализовывать для методов доступа, которые не поддерживают параллельное сканирование, или для которых объём дополнительно требующейся памяти равен нулю.

```
void
aminitparallelscan (void *target);
```

Эта функция будет вызываться для инициализации области динамической разделяемой памяти в начале параллельного сканирования. Параметр *target* будет указывать на область объёма, не

меньшего, чем возвратила функция amestimateparallelscan, и данная функция может хранить в этой области любые нужные ей данные.

Эту функцию можно не реализовывать для методов доступа, которые не поддерживают параллельное сканирование, или когда выделенная область в разделяемой памяти не требует инициализации.

```
void
amparallelrescan (IndexScanDesc scan);
```

Эта функция, если её реализовать, будет вызываться перед перезапуском параллельного сканирования индекса. Она должна сбросить всё разделяемое состояние, установленное функцией aminitparallelscan, с тем, чтобы такое сканирование перезапустилось с начала.

56.3. Сканирование индекса

В процессе сканирования индексный метод доступа отвечает только за выдачу идентификаторов всех кортежей, которые по его представлению соответствуют *ключам сканирования*. Метод доступа *не* участвует в самой процедуре выборки этих кортежей из основной таблицы и не определяет, удовлетворяют ли эти кортежи условиям видимости или другим ограничениям.

Ключом сканирования является внутреннее представление предложения WHERE в виде *ключ_индекса оператор константа*, где ключ индекса — один из столбцов индекса, а оператор — один из членов семейства операторов, связанного с типом данного столбца. При сканировании по индексу могут задаваться несколько или ноль ключей сканирования, результаты поиска которых должны неявно объединяться операцией AND — ожидается, что возвращаемые кортежи будут удовлетворять всем заданным условиям.

Метод доступа для конкретного запроса может сообщить, что индекс является *неточным* или, другими словами, требует перепроверки. Это подразумевает, что при сканировании индекса будут возвращены все записи, соответствующие ключу сканирования, плюс, возможно, дополнительные записи, которые ему не соответствуют. Внутренний механизм сканирования затем повторно применит условия индекса к кортежу данных, чтобы проверить, нужно ли его выбирать на самом деле. Если признак перепроверки не установлен, при сканировании индекса должны возвращаться только соответствующие ключам записи.

Заметьте, что именно метод доступа должен гарантировать, что корректно будут найдены все и только те записи, которые соответствуют всем переданным ключам сканирования. Также учтите, что ядро системы просто передаёт все предложения WHERE с подходящими ключами индекса и семействами операторов, не проводя семантический анализ на предмет их избыточности или противоречивости. Например, с условием WHERE x > 4 AND x > 14, где x — столбец с индексом-B-деревом, именно самой функции amrescan в методе B-дерева предоставляется возможность понять, что первый ключ сканирования избыточный и может быть отброшен. Объём предварительной обработки, которую нужно произвести для этого в amrescan, зависит от того, до какой степени метод доступа должен сводить ключи к «нормализованной» форме.

Некоторые методы доступа возвращают записи индекса в чётко определённом порядке, в отличие от других. Фактически есть два различных варианта реализации упорядоченного вывода некоторым методом доступа:

- Для методов доступа, которые всегда возвращают записи в порядке их естественной сортировки (как например, в B-дереве), устанавливается признак amcanorder. В настоящее время операторам проверки равенства и упорядочивания при этом должны назначаться номера соответствующих стратегий B-дерева.
- Для методов доступа, которые поддерживают операторы упорядочивания, устанавливается признак amcanorderbyop. Он показывает, что индекс может возвращать записи в порядке, определяемом предложением ORDER BY *ключ_индекса оператор константа*. Модификаторы для такого сканирования могут передаваться в amrescan, как описывалось ранее.

У функции amgettuple есть аргумент direction, который может принимать значение ForwardScanDirection (обычный вариант, сканирование вперёд) или BackwardScanDirection (сканирование назад). Если в первом вызове после amrescan указывается BackwardScanDirection, то множество соответствующих записей индекса сканируется от конца к началу, а не в обычном направлении от начала к концу. В этом случае amgettuple должна вернуть последний соответствующий кортеж индекса, а не первый как обычно. (Это распространяется только на методы доступа с установленным признаком amcanorder.) После первого вызова amgettuple должна быть готова продолжать сканирование в любом направлении от записи, выданной последней до этого. (Но если признак amcanbackward не установлен, при всех последующих вызовах должно сохраняться то же направление, что было в первом.)

Методы доступа, которые поддерживают упорядоченное сканирование, должны уметь «помечать» позицию сканирования и затем возвращаться к помеченной позиции (возможно, несколько раз к одной и той же позиции). Но запоминаться должна только одна позиция в ходе сканирования; последующий вызов ammarkpos переопределяет ранее сохранённую позицию. Метод доступа, не поддерживающий упорядоченное сканирование, не должен определять функции ammarkpos и amrestrpos в IndexAmRoutine; достаточно записать в эти указатели NULL.

И позиция сканирования, и отмеченная позиция (при наличии) должны поддерживаться в согласованном состоянии с учётом одновременных добавлений или удалений записей в индексе. Не будет ошибкой, если только что вставленная запись не будет выдана при сканировании, которое могло бы найти эту запись, если бы она существовала до его начала, либо если сканирование выдаст такую запись после перезапуска или возврата, даже если она не была выдана в первый раз. Подобным образом, параллельное удаление может отражаться, а может и не отражаться в результатах сканирования. Важно только, чтобы при таких операциях добавления или удаления не происходило потерь или дублирования записей, которые в этих операциях не участвовали.

Если индекс сохраняет исходные индексируемые значения данных (а не их искажённое представление), обычно полезно поддержать сканирование только индекса, при котором индекс возвращает фактические данные, а не только TID кортежа данных. Это позволит соптимизировать ввод/вывод, только если карта видимости показывает, что TID относится к полностью видимой странице; в противном случае всё равно придётся посетить кортеж, чтобы проверить его видимость для MVCC. Но это не является заботой метода доступа.

Вместо amgettuple, сканирование индекса может осуществляться функцией amgetbitmap, которая выбирает все кортежи за один вызов. Это может быть значительно эффективнее amgettuple, так как позволяет избежать циклов блокировки/разблокировки в методе доступа. В принципе, amgetbitmap должна давать тот же эффект, что и многократные вызовы amgettuple, но простоты ради мы накладываем ряд дополнительных ограничений. Во-первых, amgetbitmap возвращает все кортежи сразу и не поддерживает пометку позиций и возвращение к ним. Во-вторых, кортежи, возвращаемые в битовой карте, не упорядочиваются каким-либо определённым образом, поэтому amgetbitmap не принимает аргумент direction. (И операторы упорядочивания никогда не будут передаваться для такого сканирования.) Кроме того, сканирование только индекса с amgetbitmap неосуществимо, так как нет никакой возможности возвратить содержимое кортежей индекса. Наконец, amgetbitmap не гарантирует, что будут установлены какие-либо блокировки для возвращаемых кортежей, и следствия этого описаны в Разделе 56.4.

Заметьте, что метод доступа может реализовывать только функцию amgetbitmap, но не amgettuple, и наоборот, если его внутренняя реализация несовместима с одной из этих функций.

56.4. Замечания о блокировке с индексами

Индексные методы доступа должны справляться с параллельными операциями обновления индекса, производимыми несколькими процессами. Ядро системы Postgres Pro получает блокировку AccessShareLock для индекса в процессе сканирования и RowExclusiveLock при модификации индекса (включая и обычную очистку командой VACUUM). Так как эти типы блокировок не конфликтуют, метод доступа должен сам устанавливать более точечные блокировки, которые ему могут потребоваться. Блокировка ACCESS EXCLUSIVE индекса в целом устанавливается только при создании и уничтожении индекса или операции REINDEX.

Реализация типа индекса, поддерживающего параллельные изменения, обычно требует глубокого и всестороннего анализа требуемого поведения.

Помимо собственных внутренних требований индексов к целостности, при параллельном обновлении данных возникают вопросы согласованности родительской таблицы (*основных данных*) и индекса. Вследствие того, что Postgres Pro отделяет чтение и изменение основных данных от чтения и изменения индекса, образуются временные интервалы, в которых индекс может быть несогласованным с данными. Мы решаем эту проблему, применяя следующие правила:

- Новая запись в области данных добавляется до того, как для неё будут созданы записи в индексах. (Таким образом, при параллельном сканировании индекса эта запись в данных скорее всего не будет замечена. Это не проблема, так как читателю индекса всё равно не нужны незафиксированные строки. Но учтите написанное в Разделе 56.5.)
- Когда запись данных удаляется (командой `VACUUM`), сначала должны удалиться все созданные для неё записи в индексах.
- Сканирование индекса должно закрепить страницу индекса, на которой находится элемент, возвращённый последним вызовом `amgettuple`, а `ambulkdelete` не должна удалять записи со страниц, закреплённых другими процессами. Чем обосновано это правило, описывается ниже.

Без третьего правила читатель индекса мог бы увидеть запись индекса за мгновение до того, как она была удалена процедурой `VACUUM`, а затем обратиться к соответствующей записи данных после того, как `VACUUM` удалит и её. Это не приведёт к серьёзным проблемам, если данный элемент остаётся незадействованным, когда к нему обращается читатель, так как пустой слот будет игнорироваться функцией `heap_fetch()`. Но как быть, если третий процесс уже занял этот слот какими-то своими данными? Когда применяется снимок, совместимый с MVCC, и это не проблема, так как эти данные определённо окажутся слишком новыми при проверке видимости для данного снимка. Однако для снимка несовместимого с MVCC (например, снимка `SnapshotAny`), может так получиться, что будет возвращена строка, на самом деле не соответствующая ключам сканирования. Мы можем защититься от такого исхода, потребовав, чтобы ключи сканирования всегда перепроверялись для строки данных, но это слишком дорогостоящее решение. Вместо этого, мы закрепляем страницу индекса как промежуточный объект, показывающий, что читатель может всё ещё быть «в пути» от записи индекса к соответствующей строке данных. Благодаря тому, что `ambulkdelete` блокируется при обращении к этой закреплённой странице, процедура `VACUUM` не сможет удалить строку данных, пока её извлечение не закончит читатель. Это решение оказывается очень недорогим по времени выполнения, а издержки блокирования привносятся только в редких случаях, когда действительно возникает конфликт.

Такое решение требует, чтобы сканирования индексов выполнялись «синхронно»: мы должны выбирать каждый следующий кортеж данных сразу после того получили соответствущую запись индекса. Это оказывается невыгодно по ряду причин. «Асинхронное» сканирование, при котором мы собираем множество TID из индекса, и обращаемся за кортежами данных только после этого, влечёт гораздо меньше издержек с блокировками и позволяет обращаться к данным более эффективным образом. Согласно проведённому выше анализу, мы должны использовать синхронный подход для снимков, несовместимых с MVCC, но для запросов со снимками MVCC будет работать и асинхронное сканирование.

При сканировании индекса с `amgetbitmap`, метод доступа не закрепляет страницы индекса ни для каких из возвращаемых кортежей. Поэтому такое сканирование можно безопасно применять только со снимками MVCC.

Когда флаг `ampredlocks` не установлен, любое сканирование с данным методом доступа в сериализуемой транзакции будет получать неблокирующую предикатную блокировку для всего индекса. Это будет приводить к конфликту чтения-записи при добавлении любого кортежа в этот индекс параллельной сериализуемой транзакцией. Если среди набора параллельных сериализуемых транзакций выявляются определённые варианты конфликтов чтения-записи, одна из этих транзакций может быть отменена для сохранения целостности данных. Когда данный флаг установлен, это означает, что метод доступа реализует более точную предикатную блокировку, что способствует сокращению частоты отмены транзакций по этой причине.

56.5. Проверки уникальности в индексе

Postgres Pro реализует ограничения уникальности SQL, применяя *уникальные индексы*, то есть такие индексы, которые не принимают несколько записей с одинаковыми ключами. Для метода доступа, поддерживающего это свойство, устанавливается признак `amcanunique`. (В настоящее время это поддерживают только B-деревья.) Столбцы, указанные в предложении `INCLUDE`, не учитываются при контроле уникальности.

Вследствие особенностей MVCC, всегда необходимо допускать физическое сосуществование в индексе дублирующихся записей: такие записи могут относиться к последовательным версиям одной логической строки. На самом деле мы хотим добиться только того, чтобы никакой снимок MVCC не мог содержать две строки с одинаковыми ключами индекса. Из этого вытекают следующие ситуации, которые необходимо отследить, добавляя новую строку в уникальный индекс:

- Если конфликтующая строка была удалена текущей транзакцией, это не проблема. (В частности из-за того, что UPDATE всегда удаляет старую версию строки, прежде чем вставлять новую, операцию UPDATE можно выполнять со строкой, не меняя её ключ.)
- Если конфликтующая строка была добавлена ещё не зафиксированной транзакцией, запрос, претендующий на добавление новой строки, должен подождать, пока эта транзакция не будет зафиксирована. Если она откатывается, конфликт исчезает. Если она фиксируется и при этом оставляет конфликтующую строку, возникает нарушение уникальности. (На практике мы просто ждём завершения другой транзакции и затем пересматриваем проверку видимости.)
- Подобным образом, если конфликтующая строка была удалена ещё не зафиксированной транзакцией, запрос, претендующий на добавление новой строки, должен дождаться фиксации или отката этой транзакции, а затем повторить проверку.

Более того, непосредственно перед тем как сообщать о нарушении уникальности согласно вышеприведённым правилам, метод доступа должен перепроверить, продолжает ли существовать добавляемая строка. Если она признана «мёртвой», о предвиденном нарушении он сообщать не должен. (Такого не должно быть при обычном сценарии добавления строки текущей транзакцией, однако это может произойти в процессе `CREATE UNIQUE INDEX CONCURRENTLY`.)

Мы требуем, чтобы метод доступа выполнял эти проверки сам, и это означает, что он должен обратиться к основным данным и проверить состояние фиксации всех строк, которые согласно содержимого индекса содержат дублирующиеся ключи. Это без сомнения некрасивый и немодульный подход, но он избавляет от излишней работы: если бы мы делали отдельную пробу, то поиск конфликтующей строки по индексу пришлось бы по сути повторять, пытаясь найти место, куда вставить запись для новой строки. Более того, не представляется возможным избежать условий гонки, если проверка конфликта не будет неотъемлемой частью процедуры добавления новой записи индекса.

Если ограничение уникальности откладываемое, возникает дополнительная сложность: нам нужна возможность добавлять запись индекса для новой строки, но отложить выводы о нарушении уникальности до конца оператора или даже позже. Чтобы избежать ненужного повторного поиска по индексу, метод доступа должен произвести предварительную проверку уникальности во время изначального добавления строк. Если при этом окажется, что никакие кортежи не конфликтуют, на этом проверка заканчивается. В противном случае мы планируем перепроверку на время, когда это ограничение начинает действовать. Если во время перепроверки продолжают существовать и вставленный кортеж, и какой-либо другой с тем же ключом, должна выдаваться ошибка. (Заметьте, что в данном случае под «существованием» понимается «существование любого кортежа в цепочке HOT записей индекса».) Для реализации этой схемы в `aminsert` передаётся параметр `checkUnique`, принимающий одно из следующих значений:

- `UNIQUE_CHECK_NO` указывает, что проверка уникальности не должна выполняться (это не уникальный индекс).
- `UNIQUE_CHECK_YES` указывает, что это неоткладываемый уникальный индекс и проверку уникальности нужно выполнить немедленно, как описано выше.

- UNIQUE_CHECK_PARTIAL указывает, что это откладываемое ограничение уникальности. Postgres Pro выбирает этот режим для добавления записи индекса для каждой строки. Метод доступа должен допускать добавление в индекс дублирующихся записей и сообщать о возможных конфликтах, возвращая FALSE из aminsert. Для каждой такой строки (для которой возвращается FALSE) будет запланирована отложенная перепроверка.

  Метод доступа должен отметить все строки, которые могут нарушать ограничение уникальности, но не будет ошибкой, если он допустит ложное срабатывание. Это позволяет произвести проверку, не дожидаясь завершения других транзакций; конфликты, выявленные на этой стадии, не считаются ошибками и будут перепроверены позже, когда они могут быть уже исчерпаны.

- UNIQUE_CHECK_EXISTING указывает, что это отложенная перепроверка строки, которая была отмечена как возможно нарушающая ограничение. Хотя для этой проверки вызывается aminsert, метод доступа *не* должен добавлять новую запись индекса в данном случае, так как эта запись уже существует. Вместо этого, метод доступа должен проверить, нет ли в индексе другой такой же записи. Если она находится и соответствующая ей строка продолжает существовать, должна выдаваться ошибка.

  Для варианта UNIQUE_CHECK_EXISTING в методе доступа рекомендуется дополнительно проверить, что для целевой строки действительно имеется запись в индексе и сообщить об ошибке, если это не так. Это хорошая идея, так как значения кортежа индекса, передаваемые в aminsert, будут рассчитаны заново. Если в определении индекса задействованы функции, которые на самом деле не постоянные, мы можем проверять неправильную область индекса. Дополнительно убедившись в существовании целевой строки при перепроверке, мы можем быть уверены, что сканируются те же значения кортежа, что передавались при изначальном добавлении строки.

56.6. Функции оценки стоимости индекса

Функции amcostestimate даётся информация, описывающая возможное сканирование индекса, включая списки предложений WHERE и ORDER BY, которые были выбраны как применимые с данным индексом. Она должна вернуть оценки стоимости обращения к индексу и избирательность предложений WHERE (то есть, процент строк основной таблицы, который будет получен в ходе сканирования индекса). Для простых случаев почти всю работу оценщика стоимости можно произвести, вызывая стандартные процедуры оптимизатора; смысл существования функции amcostestimate в том, чтобы индексные методы доступа могли поделиться знаниями, специфичными для типа индекса, когда это может помочь улучшить стандартные оценки.

Каждая функция amcostestimate должна иметь такую сигнатуру:

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation,
                double *indexPages);
```

Первые три параметра передают входные значения:

root

Информация планировщика о выполняемом запросе.

path

Рассматриваемый путь доступа к индексу. В нём действительны все поля, кроме значений стоимости и избирательности.

loop_count

Число повторений сканирования индекса, которое должно приниматься во внимание при оценке стоимости. Обычно оно будет больше одного, когда при соединении со вложенным циклом планируется параметризованное сканирование. Заметьте, что оценки стоимости тем не менее должны рассчитываться для всего одного сканирования; большие значения *loop_count* лишь дают основания предположить, что при многократном сканировании положительное влияние может оказать кеширование.

Последние пять параметров — указатели на переменные для выходных значений:

**indexStartupCost*

Стоимость выполнения запуска индекса

**indexTotalCost*

Общая стоимость использования индекса

**indexSelectivity*

Избирательность индекса

**indexCorrelation*

Коэффициент корреляции между порядком сканирования индекса и порядком записей в нижележащей таблице

**indexPages*

Количество страниц индекса на уровне листьев

Заметьте, что функции оценки стоимости должны разрабатываться на C, а не на SQL или другом доступном процедурном языке, так как они должны обращаться к внутренним структурам данным планировщика/оптимизатора.

Стоимости обращения к индексу следует вычислять с использованием параметров, объявленных в `src/backend/optimizer/path/costsize.c`: последовательная выборка дискового блока имеет стоимость `seq_page_cost`, непоследовательная выборка — `random_page_cost`, а стоимость обработки одной строки индекса обычно принимается равной `cpu_index_tuple_cost`. Кроме того, за каждый оператор сравнения, вызываемый при обработке индекса, должна взиматься цена `cpu_operator_cost` (особенно за вычисление собственно условий индекса).

Стоимость доступа должна включать стоимости всех дисковых и процессорных ресурсов, требующихся для сканирования самого индекса, *но* не стоимости извлечения или обработки строк основной таблицы, с которой связан индекс.

«Стоимость запуска» составляет часть общей стоимости сканирования, которая должна быть потрачена, прежде чем можно будет начать чтение первой строки. Для большинства индексов она может считаться нулевой, но для типов индексов с высокими затратами на запуск она может быть больше нуля.

Значение в *indexSelectivity* должно показывать, какой процент строк основной таблицы ожидается получить при сканировании таблицы. В случае неточного запроса это обычно будет больше процента строк, действительно удовлетворяющих заданным ограничивающим условиям.

В *indexCorrelation* записывается корреляция (в диапазоне от -1.0 до 1.0) между порядком записей в индексе и в таблице. Это значение будет корректировать оценку стоимости выборки строк из основной таблицы.

В *indexPages* записывается число страниц на уровне листьев. Это помогает выбрать число исполнителей для параллельного сканирования индекса.

Когда *loop_count* больше нуля, возвращаться должны средние значения, ожидаемые для одного сканирования индекса.

Оценка стоимости

Типичная процедура оценки выглядит следующим образом:

1. Рассчитать и вернуть процент строк родительской таблицы, которые будут посещены при заданных ограничивающих условиях. В отсутствие каких-либо знаний, специфичных для типа индекса, использовать стандартную функцию оптимизатора `clauselist_selectivity()`:

   ```
   *indexSelectivity = clauselist_selectivity(root, path->indexquals,
                                              path->indexinfo->rel->relid,
                                              JOIN_INNER, NULL);
   ```

2. Оценить число строк индекса, которые будут посещены при сканировании. Для многих типов индексов это будет произведение *indexSelectivity* и числа строк в индексе, но оно может быть и больше. (Заметьте, что размер индекса в страницах и строках можно узнать из структуры `path->indexinfo`.)

3. Рассчитать число страниц индекса, которые будут получены при сканировании. Это может быть просто произведение *indexSelectivity* и размера индекса в страницах.

4. Вычислить стоимость обращения к индексу. Универсальный оценщик может сделать следующее:

   ```
   /*
    * Вообще предполагается, что страницы индекса будут считываться последовательно,
    * так что стоимость их чтения cost seq_page_cost, а не random_page_cost.
    * Также мы добавляем стоимость за вычисление условия индекса для каждой строки.
    * Все стоимости считаются пропорционально возрастающими при сканировании.
    */
   cost_qual_eval(&index_qual_cost, path->indexquals, root);
   *indexStartupCost = index_qual_cost.startup;
   *indexTotalCost = seq_page_cost * numIndexPages +
       (cpu_index_tuple_cost + index_qual_cost.per_tuple) * numIndexTuples;
   ```

   Однако при таком расчёте не учитывается амортизация чтения индекса при повторном сканировании.

5. Оценить корреляцию индекса. Для простого упорядоченного индекса по одному полю её можно получить из pg_statistic. Если корреляция неизвестна, вернуть консервативную оценку — ноль (корреляция отсутствует).

Примеры функций оценки стоимости можно найти в `src/backend/utils/adt/selfuncs.c`.

Глава 57. Унифицированные записи WAL

Хотя у всех внутренних модулей, взаимодействующих с WAL, имеются собственные типы записей WAL, существует также унифицированный тип записей WAL, описывающий изменения в страницах унифицированным образом. Это полезно для расширений, реализующих собственные методы доступа, так как они не могут зарегистрировать свои подпрограммы воспроизведения изменений WAL.

API для конструирования унифицированных записей WAL определён в `access/generic_xlog.h` и реализован в `access/transam/generic_xlog.c`.

Чтобы сформировать запись изменения данных для WAL, применяя механизм унифицированных записей WAL, выполните следующие действия:

1. `state = GenericXLogStart(relation)` — начните формирование унифицированной записи WAL для заданного отношения.
2. `page = GenericXLogRegisterBuffer(state, buffer, flags)` — зарегистрируйте буфер, который будет изменён текущей унифицированной записью WAL. Эта функция возвращает указатель на временную копию страницы буфера, в которой должны производиться изменения. (Модифицировать непосредственно содержимое буфера нельзя.) В третьем аргументе передаётся битовая маска флагов, применимых к этой операции. В настоящее время флаг только один — `GENERIC_XLOG_FULL_IMAGE`, который показывает, что в запись WAL нужно включить образ всей страницы, а не только изменения. Обычно этот флаг должен устанавливаться, когда страница новая или полностью перезаписана. Вызов `GenericXLogRegisterBuffer` можно повторять, если фиксируемое в WAL действие изменяет несколько страниц.
3. Примените изменения к образам страниц, полученным на предыдущем шаге.
4. `GenericXLogFinish(state)` — завершите изменения в буферах и выдайте унифицированную запись WAL.

Формирование записи WAL можно прервать на любом шаге, вызвав `GenericXLogAbort(state)`. При этом будут отменены все изменения, внесённые в копии образов страниц.

Используя механизм унифицированных записей WAL, необходимо учитывать следующее:

- Модифицировать буферы напрямую нельзя! Все изменения должны производиться в копиях, полученных от функции `GenericXLogRegisterBuffer()`. Другими словами, код, формирующий унифицированные записи WAL, никогда не должен сам вызывать `BufferGetPage()`. Однако вызывающий код отвечает за закрепление/открепление и блокировку/разблокировку буферов в подходящие моменты времени. Исключительная блокировка каждого целевого буфера должна удерживаться от вызова `GenericXLogRegisterBuffer()` до `GenericXLogFinish()`.
- Регистрацию буферов (шаг 2) и модификацию образов страниц (шаг 3) можно свободно смешивать, оба этих шага можно повторять в любой последовательности. Но помните, что буферы должны регистрироваться в том же порядке, в каком для них должны получаться блокировки при воспроизведении.
- Максимальное число буферов, которые можно зарегистрировать для унифицированной записи WAL, составляет `MAX_GENERIC_XLOG_PAGES`. При исчерпании этого лимита будет выдана ошибка.
- Унифицированный тип WAL подразумевает, что страницы, подлежащие изменению, имеют стандартную структуру, в частности между `pd_lower` и `pd_upper` нет полезных данных.
- Так как изменяются копии страниц буфера, `GenericXLogStart()` не начинает критическую секцию. Таким образом вы можете безопасно выделять память, выдать ошибку и т. п. между `GenericXLogStart()` и `GenericXLogFinish()`. Единственная фактическая критическая секция присутствует внутри `GenericXLogFinish()`. При выходе по ошибке так же не нужно заботиться о вызове `GenericXLogAbort()`.

- `GenericXLogFinish()` помечает буферы как грязные и устанавливает для них LSN. Вам делать явно это не нужно.
- Для нежурналируемых отношений всё работает так же, за исключением того, что фактически запись в WAL не выдаётся. Таким образом, явно проверять, является ли отношение нежурналируемым, не требуется.
- Функция воспроизведения унифицированных изменений WAL получит исключительные блокировки буферов в том же порядке, в каком они были зарегистрированы. После воспроизведения всех изменений блокировки в том же порядке и освобождаются.
- Если для регистрируемого буфера не задаётся `GENERIC_XLOG_FULL_IMAGE`, унифицированная запись WAL содержит различие между старым и новым образом страницы, которое вычисляется при побайтовом сравнении. Результат оказывается не очень компактным при перемещении данных в странице, но это может быть доработано в будущем.

Глава 58. Индексы GiST

58.1. Введение

GiST расшифровывается как «Generalized Search Tree» (Обобщённое поисковое дерево). Это сбалансированный иерархический метод доступа, который представляет собой базовый шаблон, на основе которого могут реализовываться произвольные схемы индексации. На базе GiST могут быть реализованы B-деревья, R-деревья и многие другие схемы индексации.

Ключевым преимуществом GiST является то, что он позволяет разрабатывать дополнительные типы данных с соответствующими методами доступа экспертам в предметной области типа данных, а не специалистам по СУБД.

Представленная здесь информация частично позаимствована с *сайта* Проекта индексации GiST Калифорнийского университета в Беркли и из диссертации Марселя Корнакера *Методы доступа для СУБД следующего поколения*. Сопровождением реализации GiST в Postgres Pro в основном занимаются Фёдор Сигаев и Олег Бартунов; дополнительные сведения можно получить на их *сайте*.

58.2. Встроенные классы операторов

В базовый дистрибутив Postgres Pro включены классы операторов GiST, перечисленные в Таблице 58.1. (Некоторые дополнительные модули, описанные в Приложении F, добавляют другие классы операторов GiST.)

Таблица 58.1. Встроенные классы операторов GiST

Имя	Индексируемый тип данных	Индексируемые операторы	Операторы упорядочивания
box_ops	box	&& &> &< &<| >> << <<| <@ @> @ |&> |>> ~ ~=	
circle_ops	circle	&& &> &< &<| >> << <<| <@ @> @ |&> |>> ~ ~=	<->
inet_ops	inet, cidr	&& >> >>= > >= <> << <<= < <= =	
point_ops	point	>> >^ << <@ <@ <@ <^ ~=	<->
poly_ops	polygon	&& &> &< &<| >> << <<| <@ @> @ |&> |>> ~ ~=	<->
range_ops	любой тип диапазона	&& &> &< >> << <@ -|- = @> @>	
tsquery_ops	tsquery	<@ @>	
tsvector_ops	tsvector	@@	

По историческим причинам класс операторов `inet_ops` не является классом по умолчанию для типов `inet` и `cidr`. Чтобы использовать его, укажите имя класса в `CREATE INDEX`, например:

```
CREATE INDEX ON my_table USING GIST (my_inet_column inet_ops);
```

58.3. Расширяемость

Реализация нового метода доступа индекса традиционно была большой и сложной задачей. Чтобы её решить, необходимо было понимать внутреннее устройство базы данных, в частности работу менеджера блокировок и журнала предзаписи. Но с интерфейсом GiST, реализующим высокий уровень абстракции, разработчик метода доступа должен реализовать только смысловое наполнение индексируемого типа данных. Уровень GiST берёт на себя заботу о параллельном доступе, поддержке журнала и поиске в структуре дерева.

Эту расширяемость не следует путать с расширяемостью других стандартных деревьев поиска в смысле поддержки различных типов данных. Например, Postgres Pro поддерживает расширяемость B-деревьев и индексов по хешу. Это означает, что в Postgres Pro вы можете построить B-дерево или хеш-таблицу по любому желаемому типу данных. Но такие B-деревья будут поддерживать только предикаты сравнений (<, =, >), а индексы по хешу только запросы с равенством.

Поэтому, если вы проиндексируете в Postgres Pro в B-дереве, например, коллекцию изображений, вы сможете выполнять только проверки вида «равны ли изображения X и Y», «меньше ли изображение X изображения Y» и «больше ли изображение X изображения Y». Это может быть полезно, в зависимости от того, как вы определите операции «равно», «меньше» и «больше». Однако используя индекс на базе GiST, возможно удовлетворять и запросы из предметной области, например, «найти все изображения лошадей» или «найти все пересвеченные изображения».

Всё, что нужно, чтобы получить работающий метод доступа GiST — это реализовать несколько методов, определяющих поведение ключей в дереве. Конечно, эти методы должны быть довольно изощрёнными, чтобы поддерживать изощрённые запросы, но для всех стандартных запросов (B-деревьев, R-деревьев и т. д.) они относительно просты. Словом, GiST сочетает расширяемость с универсальностью, повторным использованием кода и аккуратным интерфейсом.

Класс операторов индекса GiST должен предоставить семь методов и может дополнительно предоставлять ещё два. Корректность индекса обеспечивается реализацией методов `same`, `consistent` и `union`, а его эффективность (по размеру и скорости) будет зависеть от методов `penalty` и `picksplit`. Последние два основных метода `compress` и `decompress` позволяют реализовать внутреннее представление данных дерева, не совпадающее с типом индексируемых данных. Данные листьев индекса должны быть индексируемого типа, тогда как в остальных узлах дерева могут быть произвольные структуры C (но при этом должны соблюдаться правила, предъявляемые Postgres Pro к типам данных; прочитайте о `varlena` для данных переменного размера). Если внутренний тип данных дерева существует на уровне SQL, в команде `CREATE OPERATOR CLASS` можно использовать указание `STORAGE`. Необязательный восьмой метод `distance` нужно реализовать, только если класс операторов желает поддерживать упорядоченные сканирования (поиск ближайших соседей). Необязательный девятый метод `fetch` требуется, если класс операторов желает поддерживать сканирование только индекса.

`consistent`

Для переданной записи индекса `p` и значения запроса `q` эта функция определяет, является ли запись индекса «соответствующей» запросу; то есть, может ли предикат «*`индексированный_столбец индексируемый_оператор q`*» удовлетворяться для какой-либо строки, представленной данной записью индекса? Для записей на уровне листьев это равносильно проверке индексируемого условия, тогда как для внутреннего узла дерева требуется определить, нужно ли сканировать поддерево индекса, относящееся к данному узлу. Когда результат `true`, также должен возвращаться флаг `recheck`, показывающий, точно ли удовлетворяется предикат или это лишь потенциально возможно. Если `recheck = false`, это означает, что индекс проверил условие предиката в точности, тогда как при `recheck = true` проверяемая строка будет только кандидатом на совпадение. В этом случае система автоматически перепроверит *`индексируемый_оператор`* с действительным значением строки, чтобы окончательно определить, соответствует ли оно запросу. Благодаря этому GiST поддерживает индексы как точной, так и неточной структуры.

В SQL эта функция должна объявляться примерно так:

```
CREATE OR REPLACE FUNCTION my_consistent(internal, data_type, smallint, oid,
 internal)
RETURNS bool
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

А соответствующий код в модуле C может реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_consistent);
```

```
Datum
my_consistent(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type  *query = PG_GETARG_DATA_TYPE_P(1);
    StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    bool       *recheck = (bool *) PG_GETARG_POINTER(4);
    data_type  *key = DatumGetDataType(entry->key);
    bool        retval;

    /*
     * Определить возвращаемое значение как функцию стратегии, ключа и запроса.
     *
     * Вызовите GIST_LEAF(entry), чтобы узнать текущую позицию в дереве индекса,
     * что удобно, например для поддержки оператора = (вы можете проверить
     * равенство в листьях дерева и непустое пересечение в остальных
     * узлах).
     */

    *recheck = true;        /* или false, если проверка точная */

    PG_RETURN_BOOL(retval);
}
```

Здесь `key` — это элемент в индексе, а `query` — значение, искомое в индексе. Параметр `StrategyNumber` показывает, какой оператор из класса операторов применяется — он соответствует одному из номеров операторов, заданных в команде `CREATE OPERATOR CLASS`.

В зависимости от того, какие операторы включены в класс, тип данных `query` может быть разным для разных операторов, так как это будет тот тип, что фигурирует в правой части оператора, и он может отличаться от индексируемого типа данных, фигурирующего слева. (В показанном выше шаблоне предполагается, что допускается только один тип; в противном случае получение значения `query` зависело бы от оператора.) В SQL-объявлении функции `consistent` для аргумента `query` рекомендуется установить индексированный тип данного класса операторов, хотя фактический тип может быть каким-то другим, в зависимости от оператора.

`union`

Этот метод консолидирует информацию в дереве. Получив набор записей, он должен сгенерировать в индексе новую запись, представляющие все эти записи.

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_union(internal, internal)
RETURNS storage_type
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

И соответствующий код в модуле C должен реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_union);

Datum
my_union(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    GISTENTRY  *ent = entryvec->vector;
    data_type  *out,
               *tmp,
```

```
                *old;
    int          numranges,
                 i = 0;

    numranges = entryvec->n;
    tmp = DatumGetDataType(ent[0].key);
    out = tmp;

    if (numranges == 1)
    {
        out = data_type_deep_copy(tmp);

        PG_RETURN_DATA_TYPE_P(out);
    }

    for (i = 1; i < numranges; i++)
    {
        old = out;
        tmp = DatumGetDataType(ent[i].key);
        out = my_union_implementation(out, tmp);
    }

    PG_RETURN_DATA_TYPE_P(out);
}
```

Как можно заметить, в этом шаблоне мы имеем дело с типом данных, для которого `union(X, Y, Z) = union(union(X, Y), Z)`. Достаточно просто можно поддержать и такие типы данных, для которых это не выполняется, реализовав соответствующий алгоритм объединения в этом опорном методе GiST.

Результатом функции `union` должно быть значение типа хранения индекса, каким бы он ни был (он может совпадать с типом индексированного столбца, а может и отличаться от него). Функция, реализующая `union`, должна возвращать указатель на память, выделенную вызовом `palloc()`. Она не может просто вернуть полученное значение как есть, даже если оно имеет тот же тип.

Как показано выше, первый аргумент `internal` функции `union` на самом деле представляет указатель `GistEntryVector`. Во втором аргументе (его можно игнорировать) передаётся указатель на целочисленную переменную. (Раньше требовалось, чтобы функция `union` сохраняла в этой переменной размер результирующего значения, но теперь такого требования нет.)

`compress`

Преобразует элемент данных в формат, подходящий для физического хранения в странице индекса.

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_compress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

И соответствующий код в модуле C должен реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_compress);

Datum
my_compress(PG_FUNCTION_ARGS)
{
```

```
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY  *retval;

    if (entry->leafkey)
    {
        /* заменить entry->key сжатой версией */
        compressed_data_type *compressed_data =
 palloc(sizeof(compressed_data_type));

        /* заполнить *compressed_data из entry->key ... */

        retval = palloc(sizeof(GISTENTRY));
        gistentryinit(*retval, PointerGetDatum(compressed_data),
                      entry->rel, entry->page, entry->offset, FALSE);
    }
    else
    {
        /* обычно с записями внутренних узлов ничего делать не нужно */
        retval = entry;
    }

    PG_RETURN_POINTER(retval);
}
```

Разумеется, *compressed_data_type* (тип сжатых данных) нужно привести к нужному типу, при преобразовании в который будут сжиматься узлы на уровне листьев.

`decompress`

Метод, обратный к `compress`. Преобразует представление элемента данных в индексе в формат, с которым могут работать другие методы GiST в классе операторов.

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_decompress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

И соответствующий код в модуле C должен реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_decompress);

Datum
my_decompress(PG_FUNCTION_ARGS)
{
    PG_RETURN_POINTER(PG_GETARG_POINTER(0));
}
```

Этот шаблон подходит для случая, когда преобразовывать данные не нужно.

`penalty`

Возвращает значение, выражающее «стоимость» добавления новой записи в конкретную ветвь дерева. Элементы будут вставляться по тому направлению в дереве, для которого значение `penalty` минимально. Результаты `penalty` должны быть неотрицательными; если возвращается отрицательное значение, оно воспринимается как ноль.

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_penalty(internal, internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;  -- в некоторых случаях функции стоимости не должны быть строгими
```

И соответствующий код в модуле C может реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_penalty);

Datum
my_penalty(PG_FUNCTION_ARGS)
{
    GISTENTRY  *origentry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY  *newentry = (GISTENTRY *) PG_GETARG_POINTER(1);
    float      *penalty = (float *) PG_GETARG_POINTER(2);
    data_type  *orig = DatumGetDataType(origentry->key);
    data_type  *new = DatumGetDataType(newentry->key);

    *penalty = my_penalty_implementation(orig, new);
    PG_RETURN_POINTER(penalty);
}
```

По историческим причинам функция `penalty` не просто возвращает результат типа `float`; вместо этого она должна сохранить его значение по адресу, указанному третьим аргументом. Собственно возвращаемое значение игнорируется, хотя в нём принято возвращать этот же адрес.

Функция `penalty` важна для хорошей производительности индекса. Она будет вызываться во время добавления записи, чтобы выбрать ветвь для дальнейшего движения, когда в дерево нужно добавить новый элемент. Это имеет значение во время запроса, так как чем более сбалансирован индекс, тем быстрее будет поиск в нём.

`picksplit`

Когда необходимо разделить страницу индекса, эта функция решает, какие записи должны остаться в старой странице, а какие нужно перенести в новую.

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_picksplit(internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

И соответствующий код в модуле C может реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_picksplit);

Datum
my_picksplit(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    GIST_SPLITVEC *v = (GIST_SPLITVEC *) PG_GETARG_POINTER(1);
    OffsetNumber maxoff = entryvec->n - 1;
    GISTENTRY  *ent = entryvec->vector;
    int         i,
                nbytes;
    OffsetNumber *left,
                *right;
    data_type  *tmp_union;
    data_type  *unionL;
    data_type  *unionR;
    GISTENTRY **raw_entryvec;

    maxoff = entryvec->n - 1;
    nbytes = (maxoff + 1) * sizeof(OffsetNumber);
```

```
    v->spl_left = (OffsetNumber *) palloc(nbytes);
    left = v->spl_left;
    v->spl_nleft = 0;

    v->spl_right = (OffsetNumber *) palloc(nbytes);
    right = v->spl_right;
    v->spl_nright = 0;

    unionL = NULL;
    unionR = NULL;

    /* Инициализировать чистый вектор записи. */
    raw_entryvec = (GISTENTRY **) malloc(entryvec->n * sizeof(void *));
    for (i = FirstOffsetNumber; i <= maxoff; i = OffsetNumberNext(i))
        raw_entryvec[i] = &(entryvec->vector[i]);

    for (i = FirstOffsetNumber; i <= maxoff; i = OffsetNumberNext(i))
    {
        int         real_index = raw_entryvec[i] - entryvec->vector;

        tmp_union = DatumGetDataType(entryvec->vector[real_index].key);
        Assert(tmp_union != NULL);

        /*
         * Выбрать, куда помещать записи индекса и изменить unionL и unionR
         * соответственно. Добавить записи в v->spl_left или
         * v->spl_right и увеличить счётчики.
         */

        if (my_choice_is_left(unionL, curl, unionR, curr))
        {
            if (unionL == NULL)
                unionL = tmp_union;
            else
                unionL = my_union_implementation(unionL, tmp_union);

            *left = real_index;
            ++left;
            ++(v->spl_nleft);
        }
        else
        {
            /*
             * То же самое с правой стороной
             */
        }
    }

    v->spl_ldatum = DataTypeGetDatum(unionL);
    v->spl_rdatum = DataTypeGetDatum(unionR);
    PG_RETURN_POINTER(v);
}
```

Заметьте, что результат функции `picksplit` доставляется через полученную на вход структуру `v`. Собственно возвращаемое значение игнорируется, хотя в нём принято возвращать адрес `v`.

Как и `penalty`, функция `picksplit` важна для хорошей производительности индекса. Сложность создания быстродействующих индексов GiST заключается как раз в разработке подходящих реализаций `penalty` и `picksplit`.

`same`

Возвращает true, если два элемента индекса равны, и false в противном случае. («Элемент индекса» — это значение типа хранения индекса, а не обязательно исходного типа индексируемого столбца.)

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_same(storage_type, storage_type, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

И соответствующий код в модуле C может реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_same);

Datum
my_same(PG_FUNCTION_ARGS)
{
    prefix_range *v1 = PG_GETARG_PREFIX_RANGE_P(0);
    prefix_range *v2 = PG_GETARG_PREFIX_RANGE_P(1);
    bool       *result = (bool *) PG_GETARG_POINTER(2);

    *result = my_eq(v1, v2);
    PG_RETURN_POINTER(result);
}
```

По историческим причинам функция `same` не просто возвращает результат булевого типа; вместо этого она должна сохранить флаг по адресу, указанному третьим аргументом. Собственно возвращаемое значение игнорируется, хотя в нём принято возвращать этот же адрес.

`distance`

Для переданной записи индекса `p` и значения запроса `q` эта функция определяет «дистанцию» от записи индекса до значения в запросе. Эта функция должна быть представлена, если класс операторов содержит какие-либо операторы упорядочивания. Запрос с оператором упорядочивания будет выполняться так, чтобы записи индекса с наименьшей «дистанцией» возвращались первыми, так что результаты должны согласовываться со смысловым значением оператора. Для записи на уровне листьев результат представляет только дистанцию до этой записи, а для внутреннего узла дерева это будет минимальная дистанция, которая может быть получена среди всех его потомков.

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_distance(internal, data_type, smallint, oid, internal)
RETURNS float8
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

И соответствующий код в модуле C должен реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_distance);

Datum
my_distance(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type  *query = PG_GETARG_DATA_TYPE_P(1);
    StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    /* bool *recheck = (bool *) PG_GETARG_POINTER(4); */
    data_type  *key = DatumGetDataType(entry->key);
```

```
    double      retval;

    /*
     * определить возвращаемое значение как функцию стратегии, ключа и запроса.
     */

    PG_RETURN_FLOAT8(retval);
}
```

Функции `distance` передаются те же аргументы, что и функции `consistent`.

При определении дистанции допускается некоторая неточность, если результат никогда не будет превышать действительную дистанцию до элемента. Так, например, в геометрических приложениях бывает достаточно определить дистанцию до описанного прямоугольника. Для внутреннего узла дерева результат не должен превышать дистанцию до любого из его дочерних узлов. Если возвращаемая дистанция неточная, функция должна установить флаг `*recheck`. (Это необязательно для внутренних узлов дерева; для них результат всегда считается неточным.) В этом случае исполнитель вычислит точную дистанцию, выбрав кортеж из кучи, и переупорядочит кортежи при необходимости.

Если функция distance возвращает `*recheck = true` для любого узла на уровне листьев, типом результата исходного оператора упорядочивания должен быть `float8` или `float4`, и значения результата функции distance должны быть сравнимы с результатами исходного оператора упорядочивания, так как исполнитель будет выполнять сортировку, используя и результаты функции distance, и уточнённые результаты оператора упорядочивания. В противном случае значениями результата distance могут быть любые конечные значения `float8`, при условии, что относительный порядок значений результата соответствует порядку, который даёт оператор упорядочивания. (Значения бесконечность и минус бесконечность применяются внутри для особых случаев, например, представления NULL, поэтому возвращать такие значения из функций `distance` не рекомендуется.)

`fetch`

Преобразует сжатое представление элемента данных в индексе в исходный тип данных, для сканирования только индекса. Возвращаемые данные должны быть точной, не примерной копией изначально проиндексированного значения.

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_fetch(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

В качестве аргумента ей передаётся указатель на структуру `GISTENTRY`. При вызове её поле `key` содержит данные листа в сжатой форме (не NULL). Возвращаемое значение — ещё одна структура `GISTENTRY`, в которой поле `key` содержит те же данные в исходной, развёрнутой форме. Если функция compress класса операторов не делает с данными листьев ничего, метод `fetch` может возвратить аргумент без изменений.

Соответствующий код в модуле C должен реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_fetch);

Datum
my_fetch(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    input_data_type *in = DatumGetPointer(entry->key);
    fetched_data_type *fetched_data;
    GISTENTRY  *retval;
```

```
    retval = palloc(sizeof(GISTENTRY));
    fetched_data = palloc(sizeof(fetched_data_type));

    /*
     * Преобразовать структуру 'fetched_data' в Datum исходного типа данных.
     */

    /* Заполнить *retval из fetch_data. */
    gistentryinit(*retval, PointerGetDatum(converted_datum),
                  entry->rel, entry->page, entry->offset, FALSE);

    PG_RETURN_POINTER(retval);
}
```

Если метод сжатия является неточным для записей уровня листьев, такой класс операторов не может поддерживать сканирование только индекса и не должен определять функцию `fetch`.

Все опорные методы GiST обычно вызываются в кратковременных контекстах памяти; то есть, `CurrentMemoryContext` сбрасывается после обработки каждого кортежа. Таким образом можно не заботиться об освобождении любых блоков памяти, выделенных функцией palloc. Однако в некоторых случаях для опорного метода полезно кешировать какие-либо данные между вызовами. Для этого нужно разместить долгоживущие данные в контексте `fcinfo->flinfo->fn_mcxt` и сохранить указатель на них в `fcinfo->flinfo->fn_extra`. Такие данные смогут просуществовать всё время операции с индексом (например, одно сканирование индекса GiST, построение индекса или добавление кортежа в индекс). Не забудьте вызвать pfree для предыдущего значения, заменяя значение в `fn_extra`, чтобы не допустить накопления утечек памяти в ходе операции.

58.4. Реализация

58.4.1. Построение GiST с буферизацией

Если попытаться построить большой индекс GiST, просто добавляя все кортежи по очереди, скорее всего это будет медленно, потому что если кортежи индексов будут разбросаны по всему индексу, а индекс будет большим и не поместится в кеше, при добавлении записей потребуется произвести множество операций произвольного доступа. Начиная с версии 9.2, PostgreSQL поддерживает более эффективный метод построения индексов с применением буферизации, что позволяет кардинально сократить число операций произвольного доступа, требующихся при обработке неупорядоченных наборов данных. Для хорошо упорядоченных наборов выигрыш может быть минимальным или вообще отсутствовать, так как всего несколько страниц будут принимать новые кортежи в один момент времени, и эти страницы будут умещаться в кеше, даже если весь индекс очень большой.

Однако при построении индекса с буферизацией приходится гораздо чаще вызывать функцию `penalty`, на что уходят дополнительные ресурсы процессора. Кроме того, используемым для этой операции буферам требуется временное место на диске, вплоть до размера результирующего индекса. Буферизация также может повлиять на качество результирующего индекса как в положительную, так и в отрицательную сторону. Это влияние зависит от различных факторов, например от распределения поступающих данных и от реализации класса операторов.

По умолчанию при построении индекса GiST включается буферизация, когда размер индекса достигает значения effective_cache_size. Этот режим можно вручную включить или отключить с помощью параметра `buffering` команды CREATE INDEX. Поведение по умолчанию достаточно эффективно в большинстве случаев, но если входные данные упорядочены, выключив буферизацию, можно получить некоторое ускорение.

58.5. Примеры

В дистрибутив исходного кода Postgres Pro включены несколько примеров методов индексов, реализованных на базе GiST. В настоящее время ядро системы обеспечивает поддержку текстового

поиска (индексацию типов `tsvector` и `tsquery`), а также функциональность R-дерева для некоторых встроенных геометрических типов данных (см. `src/backend/access/gist/gistproc.c`). Классы операторов GiST содержатся также и в следующих дополнительных модулях (`contrib`):

`btree_gist`

Функциональность B-дерева для различных типов данных

`cube`

Индексирование для многомерных кубов

`hstore`

Модуль для хранения пар (ключ, значение)

`intarray`

RD-дерево для одномерных массивов значений int4

`ltree`

Индексирование древовидных структур

`pg_trgm`

Схожесть текста на основе статистики триграмм

`seg`

Индексирование «диапазонов чисел с плавающей точкой»

Глава 59. Индексы SP-GiST

59.1. Введение

Аббревиатура SP-GiST расшифровывается как «Space-Partitioned GiST» (GiST с разбиением пространства). SP-GiST поддерживает деревья поиска с разбиением, что облегчает разработку широкого спектра различных несбалансированных структур данных, в том числе деревьев квадрантов, а также k-мерных и префиксных деревьев. Общей характеристикой этих структур является то, что они последовательно разбивают пространство поиска на сегменты, которые не обязательно должны быть равного размера. При этом поиск, хорошо соответствующий правилу разбиения, с таким индексом может быть очень быстрым.

Эти популярные структуры данных изначально конструировались для работы в памяти. При таком применении они обычно представляются в виде набора динамически выделяемых узлов, связываемых указателями. Однако подобную схему нельзя в таком виде перенести на диск, так как цепочки указателей могут быть довольно длинными, и поэтому потребуется слишком много обращений к диску. Структуры данных для хранения на диске, напротив, должны иметь большую разветвлённость для минимизации объёма ввода/вывода. Для решения этой задачи SP-GiST сопоставляет узлы дерева поиска со страницами на диске так, чтобы при поиске требовалось обращаться только к нескольким страницам на диске, даже если при этом нужно просмотреть множество узлов.

Как и GiST, SP-GiST призван дать возможность разрабатывать дополнительные типы данных с соответствующими методами доступа экспертам в предметной области типа данных, а не специалистам по СУБД.

Представленная здесь информация частично позаимствована с *сайта* Проекта индексации SP-GiST Университета Пердью. Сопровождением реализации SP-GiST в Postgres Pro в основном занимаются Фёдор Сигаев и Олег Бартунов; дополнительные сведения можно получить на их *сайте*.

59.2. Встроенные классы операторов

В базовый дистрибутив Postgres Pro включены классы операторов SP-GiST, перечисленные в Таблице 59.1.

Таблица 59.1. Встроенные классы операторов SP-GiST

Имя	Индексируемый тип данных	Индексируемые операторы
kd_point_ops	point	<< <@ <^ >> >^ ~=
quad_point_ops	point	<< <@ <^ >> >^ ~=
range_ops	любой тип диапазона	&& &< &> -|- << <@ = >> @>
box_ops	box	<< &< && &> >> ~= @> <@ &<| <<| |>> |&>
text_ops	text	< <= = > >= ~<=~ ~<~ ~>=~ ~>~
inet_ops	inet, cidr	&& >> >>= > >= <> << <<= < <= =

Из двух классов операторов для типа point классом по умолчанию является quad_point_ops. Класс kd_point_ops поддерживает те же операторы, но использует другую структуру данных индекса, которая может дать выигрыш в скорости для некоторых приложений.

59.3. Расширяемость

SP-GiST предлагает интерфейс с высоким уровнем абстракции и таким образом требует от разработчика метода доступа реализовать только методы, специфичные для конкретного типа

данных. Ядро SP-GiST отвечает за эффективную схему обращений к диску и поиск в структуре дерева, а также берёт на себя заботу о параллельном доступе и поддержке журнала.

Кортежи в листьях дерева SP-GiST содержат значения того же типа данных, что и индексируемый столбец. На верхнем уровне эти кортежи содержат всегда исходное индексируемое значение данных, но на более нижних могут содержать только сокращённое представление, например, суффикс. В этом случае опорные функции класса операторов должны уметь восстанавливать исходное значение, собирая его из внутренних кортежей, которые нужно пройти для достижения уровня конкретного листа.

Внутренние кортежи устроены сложнее, так как они представляют собой точки разветвления в дереве поиска. Каждый внутренний кортеж содержит набор из одного или нескольких *узлов*, представляющих группы сходных значений листьев. Узел содержит ответвление, приводящее либо к другому, внутреннему кортежу нижнего уровня, либо к короткому списку кортежей в листьях, лежащих в одной странице индекса. Для каждого узла обычно задаётся *метка*, описывающая его; например, в префиксном дереве меткой может быть очередной символ в строковом значении. (С другой стороны, класс операторов может опускать метки узлов, если он имеет дело с фиксированным набором узлов во всех внутренних кортежах; см. Подраздел 59.4.2.) Дополнительно внутренний кортеж может хранить *префикс*, описывающий все его члены. В префиксном дереве это может быть общий префикс всех представленных ниже строк. Значением префикса не обязательно должен быть префикс, а могут быть любые данные, требующиеся классу операторов; например, в дереве квадрантов это может быть центральная точка, от которой отмеряются четыре квадранта. В этом случае внутренний кортеж дерева квадрантов будет также содержать четыре узла, соответствующие квадрантам вокруг этой центральной точки.

Некоторые алгоритмы деревьев требует знания уровня (или глубины) текущего кортежа, так что ядро SP-GiST даёт возможность классам операторов контролировать число уровней при спуске по дереву. Также имеется поддержка пошагового восстановления представленного значения, когда это требуется, и передачи вниз дополнительных данных (так называемых *переходящих значений*) при спуске.

> **Примечание**
>
> Ядро SP-GiST берёт на себя заботу о значениях NULL. Хотя в индексах SP-GiST не хранятся записи для NULL в индексируемых столбцах, это скрыто от кода класса операторов; записи индексов или условия поиска с NULL никогда не передаются методам класса операторов. (Предполагается, что операторы SP-GiST строгие и не могут возвращать положительный результат для значений NULL.) Поэтому значения NULL здесь больше обсуждаться не будут.

Класс операторов индекса для SP-GiST должен предоставить реализации пяти методов. Все пять методов должны по единому соглашению принимать два аргумента `internal`, первым из которых будет указатель на структуру C, содержащую входные значения для опорного метода, а вторым — указатель на структуру C, в которую должны помещаться выходные значения. Четыре из этих методов должны возвращать просто `void`, так как их результаты помещаются в выходную структуру; однако `leaf_consistent` дополнительно возвращает результат `boolean`. Эти методы не должны менять никакие поля в их входных структурах. Выходная структура всегда обнуляется перед вызовом пользовательского метода.

Пользователь должен определить следующие пять методов:

`config`

Возвращает статическую информацию о реализации индекса, включая OID типов данных префикса и метки узла.

В SQL эта функция должна объявляться так:

```
CREATE FUNCTION my_config(internal, internal) RETURNS void ...
```

В первом аргументе передаётся указатель на структуру `spgConfigIn` языка C, содержащие входные данные для функции. Во втором аргументе передаётся указатель на структуру `spgConfigOut` языка C, в которую функция должна поместить результат.

```
typedef struct spgConfigIn
{
    Oid         attType;        /* Индексируемый тип данных */
} spgConfigIn;

typedef struct spgConfigOut
{
    Oid         prefixType;     /* Тип данных префикса во внутренних кортежах */
    Oid         labelType;      /* Тип данных метки узла во внутренних кортежах */
    bool        canReturnData;  /* Класс операторов может восстановить исходные
 данные */
    bool        longValuesOK;   /* Класс может принимать значения, не умещающиеся на
 1 странице */
} spgConfigOut;
```

Поле `attType` передаётся для поддержки полиморфных классов операторов; для обычных классов операторов с фиксированным типом оно будет всегда содержать одно значение и поэтому его можно просто игнорировать.

Для классов операторов, не использующих префиксы, в `prefixType` можно установить `VOIDOID`. Подобным образом, для классов операторов, не использующих метки узлов, в `labelType` тоже можно установить `VOIDOID`. Признак `canReturnData` следует установить, если класс операторов может восстановить изначально переданное в индекс значение. Признак `longValuesOK` должен устанавливаться, только если `attType` переменной длины и класс операторов может фрагментировать длинные значения, повторяя суффиксы (см. Подраздел 59.4.1).

`choose`

Выбирает метод для добавления нового значения во внутренний кортеж.

В SQL эта функция должна объявляться так:

```
CREATE FUNCTION my_choose(internal, internal) RETURNS void ...
```

В первом аргументе передаётся указатель на структуру `spgChooseIn` языка C, содержащую входные данные для функции. Во втором аргументе передаётся указатель на структуру `spgChooseOut`, в которую функция должна поместить результат.

```
typedef struct spgChooseIn
{
    Datum       datum;          /* исходное значение, которое должно индексироваться
 */
    Datum       leafDatum;      /* текущее значение, которое должно сохраниться в
 листе */
    int         level;          /* текущий уровень (начиная с нуля) */

    /* Данные из текущего внутреннего кортежа */
    bool        allTheSame;     /* кортеж с признаком все-равны? */
    bool        hasPrefix;      /* у кортежа есть префикс? */
    Datum       prefixDatum;    /* если да, то это значение префикса */
    int         nNodes;         /* число узлов во внутреннем кортеже */
    Datum      *nodeLabels;     /* значения меток узлов (NULL, если их нет) */
} spgChooseIn;

typedef enum spgChooseResultType
{
    spgMatchNode = 1,           /* спуститься в существующий узел */
```

```
    spgAddNode,                 /* добавить узел во внутренний кортеж */
    spgSplitTuple               /* разделить внутренний кортеж (изменить его
 префикс) */
} spgChooseResultType;

typedef struct spgChooseOut
{
    spgChooseResultType resultType;     /* код действия, см. выше */
    union
    {
        struct                  /* результаты для spgMatchNode */
        {
            int         nodeN;      /* спуститься к этому узлу (нумерация с 0) */
            int         levelAdd;   /* шаг увеличения уровня */
            Datum       restDatum;  /* новое значение листа */
        }           matchNode;
        struct                  /* результаты для spgAddNode */
        {
            Datum       nodeLabel;  /* метка нового узла */
            int         nodeN;      /* куда вставлять её (нумерация с 0) */
        }           addNode;
        struct                  /* результаты для spgSplitTuple */
        {
            /* Информация для формирования нового внутреннего кортежа верхнего
 уровня с одним дочерним кортежем */
            bool        prefixHasPrefix;    /* кортеж должен иметь префикс? */
            Datum       prefixPrefixDatum;  /* если да, его значение */
            int         prefixNNodes;       /* число узлов */
            Datum      *prefixNodeLabels;   /* их метки (или NULL, если
                                             * меток нет) */
            int         childNodeN;         /* узел, который получит дочерний кортеж
 */

            /* Информация для формирования нового внутреннего кортежа нижнего уровня
 со всеми старыми узлами */
            bool        postfixHasPrefix;   /* кортеж должен иметь префикс? */
            Datum       postfixPrefixDatum; /* если да, его значение */
        }           splitTuple;
    }           result;
} spgChooseOut;
```

В `datum` задаётся исходное значение, которое должно быть вставлено в индекс. Значение `leafDatum` изначально совпадает с `datum`, но может быть другим на низких уровнях дерева, если его изменят методы `choose` или `picksplit`. Когда поиск места добавления достигает страницы уровня листа, в создаваемом кортеже листа будет сохранено текущее значение `leafDatum`. В `level` задаётся текущий уровень внутреннего кортежа, начиная с нуля для уровня корня. Признак `allTheSame` устанавливается, если текущий внутренний кортеж содержит несколько равнозначных узлов (см. Подраздел 59.4.3). Признак `hasPrefix` устанавливается, если текущий внутренний кортеж содержит префикс; в этом случае в `prefixDatum` задаётся его значение. Поле `nNodes` задаёт число дочерних узлов, содержащихся во внутреннем кортеже, а `nodeLabels` представляет массив их меток, или NULL, если меток у них нет.

Функция `choose` может определить, соответствует ли новое значение одному из существующих дочерних узлов, или что нужно добавить новый дочерний узел, или что новое значение не согласуется с префиксом кортежа и внутренний кортеж нужно разделить, чтобы получить менее ограничивающий префикс.

Если новое значение соответствует одному из существующих дочерних узлов, установите в `resultType` значение `spgMatchNode`. Установите в `nodeN` номер этого узла в массиве

узлов (нумерация начинается с нуля). Установите в `levelAdd` значение, на которое должен увеличиваться уровень (`level`) при спуске через этот узел, либо оставьте его нулевым, если класс операторов не отслеживает уровни. Установите `restDatum`, равным `datum`, если класс операторов не меняет значения данных от уровня к следующему, а в противном случае запишите в него изменённое значение, которое должно использоваться в качестве `leafDatum` на следующем уровне.

Если нужно добавить новый дочерний узел, установите в `resultType` значение `spgAddNode`. В `nodeLabel` задайте метку для нового узла, а в `nodeN` позицию (отсчитываемую от нуля), в которую должен вставляться узел в массиве узлов. После того как узел будет добавлен, функция `choose` вызывается снова с изменённым внутренним кортежем; в результате этого вызова должен быть получен результат `spgMatchNode`.

Если новое значение не согласуется с префиксом кортежа, установите в `resultType` значение `spgSplitTuple`. Это действие приводит к перемещению всех существующих узлов в новый внутренний кортеж нижнего уровня и замене существующего внутреннего кортежа кортежем, содержащим одну ссылку вниз на новый внутренний кортеж. Установите признак `prefixHasPrefix`, чтобы указать, должен ли новый верхний кортеж иметь префикс, и если да, задайте в `prefixPrefixDatum` значение префикса. Это новое значение префикса должно быть в достаточной мере менее ограничивающим, чем исходное, чтобы в индекс было принято новое значение. Запишите в `prefixNNodes` число требующихся узлов в новом кортеже, а в `prefixNodeLabels` — указатель на выделенный через palloc массив с их метками или NULL, если метки узлов не нужны. Заметьте, что общий размер нового кортежа верхнего уровня не должен превышать общий размер кортежа, который он замещает; это ограничивает длины нового префикса и новых меток. Установите в `childNodeN` индекс (начиная с нуля) узла, который будет ссылаться на новый внутренний кортеж нижнего уровня. Установите признак `postfixHasPrefix`, чтобы указать, должен ли новый внутренний кортеж нижнего уровня иметь префикс, и если да, задайте в `postfixPrefixDatum` значение префикса. Сочетание этих двух префиксов и метки узла, ссылающегося вниз, (если она есть) должно иметь то же значение, что и исходный префикс, так как нет возможности ни изменить метки узлов, перемещённых в новый кортеж нижнего уровня, ни изменить какие-либо нижние записи индекса. После того как узел разделён, функция `choose` будет вызвана снова с заменяемым внутренним кортежем. При этом вызове может быть возвращён результат `spgAddNode`, если подходящий узел не был создан действием `spgSplitTuple`. В конце концов `choose` должна вернуть `spgMatchNode`, чтобы операция добавления могла перейти на следующий уровень.

`picksplit`

Выбирает, как создать новый внутренний кортеж по набору кортежей в листьях.

В SQL эта функция должна объявляться так:

```
CREATE FUNCTION my_picksplit(internal, internal) RETURNS void ...
```

В первом аргументе передаётся указатель на структуру `spgPickSplitIn` языка C, содержащую входные данные для функции. Во втором аргументе передаётся указатель на структуру `spgPickSplitOut` языка C, в которую функция должна поместить результат.

```
typedef struct spgPickSplitIn
{
    int          nTuples;        /* число кортежей в листьях */
    Datum       *datums;         /* их значения (массив длины nTuples) */
    int          level;          /* текущий уровень (отсчитывая от 0) */
} spgPickSplitIn;

typedef struct spgPickSplitOut
{
    bool         hasPrefix;      /* новый внутренний кортеж должен иметь префикс? */
    Datum        prefixDatum;    /* если да, его значение */
```

```
    int         nNodes;         /* число узлов для нового внутреннего кортежа */
    Datum      *nodeLabels;     /* их метки (или NULL, если их нет) */

    int        *mapTuplesToNodes;   /* номер узла для каждого кортежа в листе */
    Datum      *leafTupleDatums;    /* значения, помещаемые в каждый новый кортеж */
} spgPickSplitOut;
```

В `nTuples` задаётся число предоставленных кортежей уровня листьев, а `datums` — массив их значений данных. В `level` указывается текущий уровень, который должны разделять все кортежи листьев, и который станет уровнем нового внутреннего кортежа.

Установите признак `hasPrefix`, чтобы указать, должен ли новый внутренний кортеж иметь префикс, и если да, задайте в `prefixDatum` значение префикса. Установите в `nNodes` количество узлов, которые будут содержаться во внутреннем кортеже, а в `nodeLabels` — массив значений их меток либо NULL, если узлам не нужны метки. Поместите в `mapTuplesToNodes` указатель на массив, назначающий номера узлов (начиная с нуля) каждому кортежу листа. В `leafTupleDatums` передайте массив значений, которые должны быть сохранены в новых кортежах листьев (они будут совпадать со входными значениями (`datums`), если класс операторов не изменяет значения от уровня к следующему). Заметьте, что функция `picksplit` сама должна выделить память, используя palloc, для массивов `nodeLabels`, `mapTuplesToNodes` и `leafTupleDatums`.

Если передаётся несколько кортежей листьев, ожидается, что функция `picksplit` классифицирует их и разделит на несколько узлов; иначе нельзя будет разнести кортежи листьев по разным страницам, что является конечной целью этой операции. Таким образом, если `picksplit` в итоге помещает все кортежи листьев в один узел, ядро SP-GiST меняет это решение и создаёт внутренний кортеж, в котором кортежи листьев связываются случайным образом с несколькими узлами с одинаковыми метками. Такой кортеж помечается флагом `allTheSame`, показывающим, что все узлы равны. Функции `choose` и `inner_consistent` должны работать с такими внутренними кортежами особым образом. За дополнительными сведениями обратитесь к Подразделу 59.4.3.

`picksplit` может применяться к одному кортежу на уровне листьев, только когда функция `config` установила в `longValuesOK` значение true и было передано входное значение, большее страницы. В этом случае цель операции — отделить префикс и получить новое, более короткое значение для листа. Этот вызов будет повторяться, пока значение уровня листа не уменьшится настолько, чтобы уместиться в странице. За дополнительными сведениями обратитесь к Подразделу 59.4.1.

`inner_consistent`

Возвращает набор узлов (ветвей), по которым надо продолжать поиск.

В SQL эта функция должна объявляться так:

```
CREATE FUNCTION my_inner_consistent(internal, internal) RETURNS void ...
```

В первом аргументе передаётся указатель на структуру `spgInnerConsistentIn` языка C, содержащую входные данные для функции. Во втором аргументе передаётся указатель на структуру `spgInnerConsistentOut` языка C, в которую функция должна поместить результат.

```
typedef struct spgInnerConsistentIn
{
    ScanKey     scankeys;       /* массив операторов и искомых значений */
    int         nkeys;          /* длина массива */

    Datum       reconstructedValue;     /* значение, восстановленное для родителя */
    void       *traversalValue; /* переходящее значение, специфичное для класса
 операторов */
    MemoryContext traversalMemoryContext;   /* переходящие значения нужно помещать
 сюда */
```

```
    int         level;          /* текущий уровень (отсчитывается от нуля) */
    bool        returnData;     /* нужно ли возвращать исходные данные? */

    /* Данные из текущего внутреннего кортежа */
    bool        allTheSame;     /* кортеж с признаком все-равны? */
    bool        hasPrefix;      /* у кортежа есть префикс? */
    Datum       prefixDatum;    /* если да, то это значение префикса */
    int         nNodes;         /* число узлов во внутреннем кортеже */
    Datum      *nodeLabels;     /* значения меток узлов (NULL, если их нет) */
} spgInnerConsistentIn;

typedef struct spgInnerConsistentOut
{
    int         nNodes;         /* число дочерних узлов, которые нужно посетить */
    int        *nodeNumbers;    /* их номера в массиве узлов */
    int        *levelAdds;      /* шаги увеличения уровня для этих узлов */
    Datum      *reconstructedValues;    /* связанные восстановленные значения */
    void      **traversalValues;        /* переходящие значения, специфичные для
 класса операторов */
} spgInnerConsistentOut;
```

Массив `scankeys` длины `nkeys` описывает условия поиска по индексу. Эти условия объединяются операцией И — найдены должны быть только те записи, которые удовлетворяют всем условиям. (Заметьте, что с `nkeys` = 0 подразумевается, что запросу удовлетворяют все записи в индексе.) Обычно эту функцию интересуют только поля `sk_strategy` и `sk_argument` в каждой записи массива, в которых определяется соответственно индексируемый оператор и искомое значение. В частности, нет необходимости проверять `sk_flags`, чтобы распознать NULL в искомом значении, так как ядро SP-GiST отфильтрует такие условия. В `reconstructedValue` передаётся значение, восстановленное для родительского кортежа; это может быть `(Datum) 0` на уровне корня или если функция `inner_consistent` не установила значение на предыдущем уровне. В `traversalValue` передаётся указатель на переходящие данные, полученные из предыдущего вызова `inner_consistent` для родительского кортежа индекса, либо NULL на уровне корня. Поле `traversalMemoryContext` указывает на контекст памяти, в котором нужно сохранить выходные переходящие данные (см. ниже). В `level` передаётся уровень текущего внутреннего кортежа (уровень корня считается нулевым). Флаг `returnData` устанавливается, когда для этого запроса нужно получить восстановленные данные; это возможно, только если функция `config` установила признак `canReturnData`. Признак `allTheSame` устанавливается, если текущий внутренний кортеж имеет пометку «все-равны»; в этом случае все узлы имеют одну метку (если имеют) и значит, либо все они, либо никакой не соответствует запросу (см. Подраздел 59.4.3). Признак `hasPrefix` устанавливается, если текущий внутренний кортеж содержит префикс; в этом случае в `prefixDatum` находится его значение. В `nNodes` задаётся число дочерних узлов, содержащихся во внутреннем кортеже, а в `nodeLabels` — массив их меток либо NULL, если они не имеют меток.

В `nNodes` нужно записать число дочерних узлов, которые потребуется посетить при поиске, а в `nodeNumbers` — массив их индексов. Если класс операторов отслеживает уровни, в `levelAdds` нужно передать массив с шагами увеличения уровня при посещении каждого узла. (Часто шаг будет одним для всех узлов, но может быть и по-другому, поэтому применяется массив.) Если потребовалось восстановить значения, поместите в `reconstructedValues` указатель на массив значений, восстановленных для каждого дочернего узла, который нужно посетить; в противном случае оставьте `reconstructedValues` равным NULL. Если желательно передать дополнительные данные («переходящие значения») на нижние уровни при поиске по дереву, поместите в `traversalValues` указатель на массив соответствующих переходящих значений, по одному для каждого дочернего узла, который нужно посетить; в противном случае оставьте в `traversalValues` значение NULL. Заметьте, что функция `inner_consistent` сама должна выделять память, используя palloc, для массивов `nodeNumbers`, `levelAdds`, `distances`, `reconstructedValues` и `traversalValues` в текущем контексте памяти. Однако выходные переходящие значения, на которые указывает массив `traversalValues`, должны размещаться

в контексте `traversalMemoryContext`. При этом каждое переходящее значения должно располагаться в отдельном блоке памяти palloc.

`leaf_consistent`

Возвращает true, если кортеж листа удовлетворяет запросу.

В SQL эта функция должна объявляться так:

```
CREATE FUNCTION my_leaf_consistent(internal, internal) RETURNS bool ...
```

В первом аргументе передаётся указатель на структуру `spgLeafConsistentIn` языка C, содержащую входные данные для функции. Во втором аргументе передаётся указатель на структуру `spgLeafConsistentOut` языка C, в которую функция должна поместить результат.

```
typedef struct spgLeafConsistentIn
{
    ScanKey     scankeys;       /* массив операторов и искомых значений */
    int         nkeys;          /* длина массива */

    Datum       reconstructedValue;     /* значение, восстановленное для родителя */
    void       *traversalValue; /* переходящее значение, специфичное для класса
 операторов */
    int         level;          /* текущий уровень (отсчитывая от нуля) */
    bool        returnData;     /* нужно ли возвращать исходные данные? */

    Datum       leafDatum;      /* значение в кортеже листа */
} spgLeafConsistentIn;

typedef struct spgLeafConsistentOut
{
    Datum       leafValue;      /* восстановленные исходные данные, при наличии */
    bool        recheck;        /* true, если оператор нужно перепроверить */
} spgLeafConsistentOut;
```

Массив `scankeys` длины `nkeys` описывает условия поиска по индексу. Эти условия объединяются операцией И — запросу удовлетворяют только те записи в индексе, которые удовлетворяют всем этим условиям. (Заметьте, что с `nkeys` = 0 подразумевается, что запросу удовлетворяют все записи в индексе.) Обычно эту функцию интересуют только поля `sk_strategy` и `sk_argument` в каждой записи массива, в которых определяются соответственно индексируемый оператор и искомое значение. В частности, нет необходимости проверять `sk_flags`, чтобы распознать NULL в искомом значении, так как ядро SP-GiST отфильтрует такие условия. В `reconstructedValue` передаётся значение, восстановленное для родительского кортежа; это может быть `(Datum) 0` на уровне корня или если функция `inner_consistent` не установила значение на предыдущем уровне. В `traversalValue` передаётся указатель на переходящие данные, полученные из предыдущего вызова `inner_consistent` для родительского кортежа индекса, либо NULL на уровне корня. В `level` передаётся уровень текущего внутреннего кортежа (уровень корня считается нулевым). Флаг `returnData` устанавливается, когда для этого запроса нужно получить восстановленные данные; это возможно, только если функция `config` установила признак `canReturnData`. В `leafDatum` передаётся значение ключа, записанное в текущем кортеже листа.

Эта функция должна вернуть `true`, если кортеж листа соответствует запросу, или `false` в противном случае. В случае положительного результата, если в поле `returnData` передано `true`, нужно поместить в `leafValue` значение, изначально переданное для индексации в этот кортеж. Кроме того, флагу `recheck` можно присвоить `true`, если соответствие неточное, так что для установления точного результата проверки нужно повторно применить оператор(ы) к актуальному кортежу данных.

Все опорные методы SP-GiST обычно вызываются в кратковременных контекстах памяти; то есть `CurrentMemoryContext` сбрасывается после обработки каждого кортежа. Таким образом, можно не

заботиться об освобождении любых блоков памяти, выделенных функцией palloc. (Метод `config` является исключением: в нём нужно не допускать утечек памяти. Но обычно метод `config` не делает ничего, кроме как присваивает константы переданной структуре параметров.)

Если индексируемый столбец имеет сортируемый тип данных, правило сортировки индекса будет передаваться всем опорным методам, используя стандартный механизм `PG_GET_COLLATION()`.

59.4. Реализация

В этом разделе освещаются тонкости реализации и особенности, о которых полезно знать тем, кто будет реализовывать классы операторов SP-GiST.

59.4.1. Ограничения SP-GiST

Отдельные кортежи листьев и внутренние кортежи должны умещаться в одной странице индекса (по умолчанию её размер 8 Кбайт). Таким образом при индексировании значений типов данных переменной длины большие значения могут поддерживаться только такими схемами, как префиксные деревья, в которых каждый уровень дерева включает префикс, достаточно короткий для помещения в страницу, и на конечном уровне листьев содержится суффикс, который также достаточно мал, чтобы поместиться в странице. Класс операторов должен устанавливать признак `longValuesOK`, только если он готов организовывать такую структуру. Если этот признак не установлен, ядро SP-GiST не примет запрос на индексацию значения, которое слишком велико для одной страницы индекса.

Также класс операторов должен отвечать за то, чтобы внутренние кортежи при расширении не выходили за пределы страницы индекса; это ограничивает число дочерних узлов, которые могут принадлежать одному внутреннему кортежу, а также максимальный размер значения префикса.

Ещё одно ограничение состоит в том, что когда узел внутреннего кортежа указывает на набор кортежей листьев, все эти кортежи должны находиться в одной странице индекса. (Это конструктивное ограничение введено для оптимизации позиционирования и экономии места на ссылках, связывающих такие кортежи вместе.) Если набор кортежей листьев оказывается слишком большим для одной страницы, выполняется разделение и вставляется промежуточный внутренний кортеж. Чтобы устранить возникшую проблему, новый внутренний кортеж *должен* разделять набор значений в листе на несколько групп узлов. Если функция `picksplit` класса операторов не может сделать это, ядро SP-GiST переходит к чрезвычайным мерам, описанным в Подразделе 59.4.3.

Когда `longValuesOK` имеет значение true, ожидается, что последующие уровни дерева SP-GiST будут включать всё больше и больше информации в префиксы и метки узлов внутренних кортежей, постепенно уменьшая значение, которое нужно сохранить на уровне листьев, чтобы оно в конце концов уместилось на странице. Чтобы в случае ошибки в классах операторах исключить бесконечные циклы при добавлении записи, ядро SP-GiST выдаст ошибку, если значение для листа не уменьшится за 10 вызовов метода `choose`.

59.4.2. SP-GiST без меток узлов

В некоторых древовидных схемах каждый внутренний кортеж содержит фиксированный набор узлов; например, в дереве квадрантов это всегда четыре узла, соответствующие четырём квадрантам вокруг центральной точки внутреннего кортежа. В таком случае код обычно работает с узлами по номерам и необходимости в явных метках узлов нет. Чтобы убрать метки узлов (и таким образом сэкономить место), функция `picksplit` может возвратить NULL вместо массива `nodeLabels`, а функция `choose` аналогично может возвратить NULL вместо массива `prefixNodeLabels` во время действия `spgSplitTuple`. В результате при последующих вызовах функций `choose` и `inner_consistent` им вместо `nodeLabels` будет передаваться NULL. В принципе метки узлов могут применяться для одних внутренних кортежей и отсутствовать у других в том же индексе.

Когда внутренний кортеж содержит узлы без меток, функция `choose` не может выбрать действие `spgAddNode`, так как в этом случае предполагается, что набор узлов фиксированный.

59.4.3. Внутренние кортежи «все-равны»

Ядро SP-GiST может переопределить результаты функции `picksplit` класса операторов, когда эта функция не может разделить поступившие значения листьев на минимум две категории узлов. Когда это происходит, создаётся новый внутренний кортеж с несколькими узлами, каждый из которых имеет одну метку (если имеет), которую `picksplit` дала одному узлу, а значения листьев распределяются случайно между этими равнозначными узлами. Для этого внутреннего кортежа устанавливается флаг `allTheSame`, который предупреждает функции `choose` и `inner_consistent`, что кортеж не содержит набор узлов, который они обычно ожидают.

Когда обрабатывается кортеж с флагом `allTheSame`, выбранное функцией `choose` действие `spgMatchNode` воспринимается как указание, что новое значение можно присвоить одному из равнозначных узлов; код ядра будет игнорировать полученное значение `nodeN` и спустится в один из узлов, выбранный случайно (чтобы дерево было сбалансированным). Будет считаться ошибкой, если `choose` выберет действие `spgAddNode`, так как при этом не все узлы окажутся равны; если добавляемое значение не соответствует существующим узлам, должно выбираться действие `spgSplitTuple`.

Также, когда обрабатывается кортеж с флагом `allTheSame`, функция `inner_consistent` должна вернуть все или не возвращать никакие узлы для продолжения поиска по индексу, так как все узлы равнозначны. Для этого может потребоваться, а может и не потребоваться код обработки особого случая, в зависимости от того, как `inner_consistent` обычно воспринимает узлы.

59.5. Примеры

Дистрибутив исходного кода Postgres Pro содержит несколько примеров классов операторов индекса SP-GiST, перечисленных в Таблице 59.1. Код их реализации вы можете найти в `src/backend/access/spgist/` и `src/backend/utils/adt/`.

Глава 60. Индексы GIN

60.1. Введение

GIN расшифровывается как «Generalized Inverted Index» (Обобщённый инвертированный индекс). GIN предназначается для случаев, когда индексируемые значения являются составными, а запросы, на обработку которых рассчитан индекс, ищут значения элементов в этих составных объектах. Например, такими объектами могут быть документы, а запросы могут выполнять поиск документов, содержащих определённые слова.

Здесь мы используем термин *объект*, говоря о составном значении, которое индексируется, и термин *ключ*, говоря о включённом в него элементе. GIN всегда хранит и ищет ключи, а не объекты как таковые.

Индекс GIN сохраняет набор пар (ключ, список идентификаторов), где *список идентификаторов* содержит идентификаторы строк, в которых находится ключ. Один и тот же идентификатор строки может фигурировать в нескольких списках, так как объект может содержать больше одного ключа. Значение каждого ключа хранится только один раз, так что индекс GIN очень компактен в случаях, когда один ключ встречается много раз.

GIN является обобщённым в том смысле, что код метода доступа GIN не должен знать о конкретных операциях, которые он ускоряет. Вместо этого задаются специальные стратегии для конкретных типов данных. Стратегия определяет, как извлекаются ключи из индексируемых объектов и условий запросов, и как установить, действительно ли удовлетворяет запросу строка, содержащая некоторые значения ключей.

Ключевым преимуществом GIN является то, что он позволяет разрабатывать дополнительные типы данных с соответствующими методами доступа экспертам в предметной области типа данных, а не специалистам по СУБД. В этом аспекте он похож на GiST.

Сопровождением реализации GIN в Postgres Pro в основном занимаются Фёдор Сигаев и Олег Бартунов. Дополнительные сведения о GIN можно получить на их *сайте*.

60.2. Встроенные классы операторов

В базовый дистрибутив Postgres Pro включены классы операторов GIN, перечисленные в Таблице 60.1. (Некоторые дополнительные модули, описанные в Приложении F, добавляют другие классы операторов GIN.)

Таблица 60.1. Встроенные классы операторов GIN

Имя	Индексируемый тип данных	Индексируемые операторы
`array_ops`	`anyarray`	`&& <@ = @>`
`jsonb_ops`	`jsonb`	`? ?& ?| @>`
`jsonb_path_ops`	`jsonb`	`@>`
`tsvector_ops`	`tsvector`	`@@ @@@`

Из двух классов операторов для типа `jsonb` классом по умолчанию является `jsonb_ops`. Класс `jsonb_path_ops` поддерживает меньше операторов, но обеспечивает для них большую производительность. За подробностями обратитесь к Подразделу 8.14.4.

60.3. Расширяемость

Интерфейс GIN характеризуется высоким уровнем абстракции и таким образом требует от разработчика метода доступа реализовать только смысловое наполнение обрабатываемого типа

данных. Уровень GIN берёт на себя заботу о параллельном доступе, поддержке журнала и поиске в структуре дерева.

Всё, что нужно, чтобы получить работающий метод доступа GIN — это реализовать несколько пользовательских методов, определяющих поведение ключей в дереве и отношения между ключами, индексируемыми объектами и поддерживаемыми запросами. Словом, GIN сочетает расширяемость с универсальностью, повторным использованием кода и аккуратным интерфейсом.

Класс операторов должен предоставить GIN следующие два метода:

`Datum *extractValue(Datum itemValue, int32 *nkeys, bool **nullFlags)`

Возвращает массив ключей (выделенный через palloc) для индексируемого объекта. Число возвращаемых ключей должно записываться в `*nkeys`. Если какой-либо из ключей может быть NULL, нужно так же выделить через palloc массив из `*nkeys` полей `bool`, записать его адрес в `*nullFlags` и установить эти флаги NULL как требуется. В `*nullFlags` можно оставить значение `NULL` (это начальное значение), если все ключи отличны от NULL. Эта функция может возвратить `NULL`, если объект не содержит ключей.

`Datum *extractQuery(Datum query, int32 *nkeys, StrategyNumber n, bool **pmatch, Pointer **extra_data, bool **nullFlags, int32 *searchMode)`

Возвращает массив ключей (выделенный через palloc) для запрашиваемого значения; то есть, в `query` поступает значение, находящееся по правую сторону индексируемого оператора, по левую сторону которого указан индексируемый столбец. Аргумент `n` задаёт номер стратегии оператора в классе операторов (см. Подраздел 35.14.2). Часто функция `extractQuery` должна проанализировать `n`, чтобы определить тип данных аргумента `query` и выбрать метод для извлечения значений ключей. Число возвращаемых ключей должно быть записано в `*nkeys`. Если какие-либо ключи могут быть NULL, нужно так же выделить через palloc массив из `*nkeys` полей `bool`, сохранить его адрес в `*nullFlags`, и установить эти флаги NULL как требуется. В `*nullFlags` можно оставить значение `NULL` (это начальное значение), если все ключи отличны от NULL. Эта функция может возвратить `NULL`, если `query` не содержит ключей.

Выходной аргумент `searchMode` позволяет функции `extractQuery` выбрать режим, в котором должен выполняться поиск. Если `*searchMode` имеет значение `GIN_SEARCH_MODE_DEFAULT` (это значение устанавливается перед вызовом), подходящими кандидатами считаются только те объекты, которые соответствуют минимум одному из возвращённых ключей. Если в `*searchMode` установлено значение `GIN_SEARCH_MODE_INCLUDE_EMPTY`, то в дополнение к объектам с минимум одним совпадением ключа, подходящими кандидатами будут считаться и объекты, вообще не содержащие ключей. (Этот режим полезен для реализации, например, операторов A-является-подмножеством-B.) Если в `*searchMode` установлено значение `GIN_SEARCH_MODE_ALL`, подходящими кандидатами считаются все отличные от NULL объекты в индексе, независимо от того, встречаются ли в них возвращаемые ключи. (Этот режим намного медленнее двух других, так как он по сути требует сканирования всего индекса, но он может быть необходим для корректной обработки крайних случаев. Оператор, который выбирает этот режим в большинстве ситуаций, скорее всего не подходит для реализации в классе операторов GIN.) Символы для этих значений режима определены в `access/gin.h`.

Выходной аргумент `pmatch` используется, когда поддерживается частичное соответствие. Чтобы использовать его, `extractQuery` должна выделить массив из `*nkeys` логических элементов и сохранить его адрес в `*pmatch`. Элемент этого массива должен содержать TRUE, если соответствующий ключ требует частичного соответствия, и FALSE в противном случае. Если переменная `*pmatch` содержит `NULL`, GIN полагает, что частичное соответствие не требуется. В эту переменную записывается `NULL` перед вызовом, так что этот аргумент можно просто игнорировать в классах операторов, не поддерживающих частичное соответствие.

Выходной аргумент `extra_data` позволяет функции `extractQuery` передать дополнительные данные методам `consistent` и `comparePartial`. Чтобы использовать его, `extractQuery` должна выделить массив из `*nkeys` указателей и сохранить его адрес в `*extra_data`, а затем сохранить всё, что ей требуется, в отдельных указателях. В эту переменную записывается `NULL` перед

вызовом, поэтому данный аргумент может просто игнорироваться классами операторов, которым не нужны дополнительные данные. Если массив *extra_data задан, он целиком передаётся в метод consistent, а в comparePartial передаётся соответствующий его элемент.

Класс операторов должен также предоставить функцию для проверки, соответствует ли индексированный объект запросу. Поддерживаются две её вариации: логическая consistent и троичная triConsistent. Функция triConsistent покрывает функциональность обоих, так что достаточно реализовать только её. Однако если вычисление логической вариации оказывается значительно дешевле, может иметь смысл реализовать их обе. Если представлена только логическая вариация, некоторые оптимизации, построенные на отбраковывании объектов до выборки всех ключей, отключаются.

bool consistent(bool check[], StrategyNumber n, Datum query, int32 nkeys, Pointer extra_data[], bool *recheck, Datum queryKeys[], bool nullFlags[])

Возвращает TRUE, если индексированный объект удовлетворяет оператору запроса с номером стратегии n (или потенциально удовлетворяет, когда возвращается указание перепроверки). Эта функция не имеет прямого доступа к значению индексированного объекта, так как GIN не хранит сами объекты. Вместо этого, она знает о значениях ключей, извлечённых из запроса и встречающихся в данном индексированном объекте. Массив check имеет длину nkeys, что равняется числу ключей, ранее возвращённых функцией extractQuery для данного значения query. Элемент массива check равняется TRUE, если индексированный объект содержит соответствующий ключ запроса; то есть, если (check[i] == TRUE), то i-ый ключ в массиве результата extractQuery присутствует в индексированном объекте. Исходное значение query передаётся на случай, если оно потребуется методу consistent; с той же целью ему передаются массивы queryKeys[] и nullFlags[], ранее возвращённые функцией extractQuery. В аргументе extra_data передаётся массив дополнительных данных, возвращённый функцией extractQuery, или NULL, если дополнительных данных нет.

Когда extractQuery возвращает ключ NULL в queryKeys[], соответствующий элемент check[] содержит TRUE, если индексированный объект содержит ключ NULL; то есть можно считать, что элементы check[] отражают условие IS NOT DISTINCT FROM. Функция consistent может проверить соответствующий элемент nullFlags[], если ей нужно различать соответствие с обычным значением и соответствие с NULL.

В случае успеха в *recheck нужно записать TRUE, если кортеж данных нужно перепроверить с оператором запроса, либо FALSE, если проверка по индексу была точной. То есть результат FALSE гарантирует, что кортеж данных не соответствует запросу; результат TRUE со значением *recheck, равным FALSE, гарантирует, что кортеж данных соответствует запросу; а результат TRUE со значением *recheck, равным TRUE, означает, что кортеж данных может соответствовать запросу, поэтому его нужно выбрать и перепроверить, применив оператор запроса непосредственно к исходному индексированному элементу.

GinTernaryValue triConsistent(GinTernaryValue check[], StrategyNumber n, Datum query, int32 nkeys, Pointer extra_data[], Datum queryKeys[], bool nullFlags[])

Функция triConsistent подобна consistent, но вместо логических значений в векторе check ей передаются три варианта сравнений для каждого ключа: GIN_TRUE, GIN_FALSE и GIN_MAYBE. GIN_FALSE и GIN_TRUE имеют обычное логическое значение, тогда как GIN_MAYBE означает, что присутствие ключа неизвестно. Когда присутствуют значения GIN_MAYBE, функция должна возвращать GIN_TRUE, только если объект удовлетворяет запросу независимо от того, содержит ли индекс соответствующие ключи запроса. Подобным образом, функция должна возвращать GIN_FALSE, только если объект не удовлетворяет запросу независимо от того, содержит ли он ключи GIN_MAYBE. Если результат зависит от элементов GIN_MAYBE, то есть соответствие нельзя утверждать или отрицать в зависимости от известных ключей запроса, функция должна вернуть GIN_MAYBE.

Когда в векторе check нет элементов GIN_MAYBE, возвращаемое значение GIN_MAYBE равнозначно установленному флагу recheck в логической функции consistent.

Кроме того, GIN нужно каким-то образом сортировать значения ключа, хранящиеся в индексе. Класс операторов может определить порядок сортировки, указав метод сравнения:

`int compare(Datum a, Datum b)`

Сравнивает два ключа (не индексированные объекты!) и возвращает целое меньше нуля, ноль или целое больше нуля, показывающее, что первый ключ меньше, равен или больше второго. Ключи NULL никогда не передаются этой функции.

Если же класс операторов не определяет метод `compare`, GIN попытается найти класс операторов B-дерева по умолчанию для типа данных ключа индекса и воспользоваться его функцией сравнения. Если класс операторов GIN предназначен только для одного типа данных, рекомендуется задавать функцию сравнения в этом классе операторов, так как поиск класса операторов B-дерева занимает несколько циклов. Однако для полиморфных классов операторов GIN (например, `array_ops`) задать одну функцию сравнения обычно не представляется возможным.

Дополнительно класс операторов для GIN может предоставить следующий метод:

`int comparePartial(Datum partial_key, Datum key, StrategyNumber n, Pointer extra_data)`

Сравнивает ключ запроса с частичным соответствием с ключом индекса. Возвращает целое число, знак которого отражает результат сравнения: отрицательное число означает, что ключ индекса не соответствует запросу, но нужно продолжать сканирование индекса; ноль означает, что ключ индекса соответствует запросу; положительное число означает, что сканирование индекса нужно прекратить, так как других соответствий не будет. Функции передаётся номер стратегии `n` оператора, сформировавшего запрос частичного соответствия, на случай, если нужно знать его смысл, чтобы определить, когда прекращать сканирование. Кроме того, ей передаётся `extra_data` — соответствующий элемент массива дополнительных данных, сформированного функцией `extractQuery`, либо `NULL`, если дополнительных данных нет. Значения NULL этой функции никогда не передаются.

Для поддержки проверок на «частичное соответствие» класс операторов должен предоставить метод `comparePartial`, а метод `extractQuery` должен устанавливать параметр `pmatch`, когда встречается запрос на частичное соответствие. За подробностями обратитесь к Подразделу 60.4.2.

Фактические типы данных различных значений `Datum`, упоминаемых выше, зависят от класса операторов. Значения объектов, передаваемые в `extractValue`, всегда имеют входной тип класса операторов, а все значения ключей должны быть типа, заданного параметром `STORAGE` для класса. Типом аргумента `query`, передаваемого функциям `extractQuery`, `consistent` и `triConsistent`, будет тот тип, что указан в качестве типа правого операнда оператора-члена класса, определяемого по номеру стратегии. Это не обязательно должен быть индексируемый тип, достаточно лишь, чтобы из него можно было извлечь значения ключей, имеющие нужный тип. Однако рекомендуется, чтобы в SQL-объявлениях этих трёх опорных функций для аргумента `query` назначался индексируемый тип класса операторов, даже несмотря на то, что фактический тип может быть другим, в зависимости от оператора.

60.4. Реализация

Внутри индекс GIN содержит B-дерево, построенное по ключам, где каждый ключ является элементом одного или нескольких индексированных объектов (например, членом массива) и где каждый кортеж на страницах листьев содержит либо указатель на B-дерево указателей на данные («дерево идентификаторов»), либо простой список указателей на данные («список идентификаторов»), когда этот список достаточно мал, чтобы уместиться в одном кортеже индекса вместе со значением ключа.

Начиная с PostgreSQL версии 9.1, в индекс могут быть включены значения ключей, равные NULL. Кроме того, в индекс вставляются NULL для индексируемых объектов, равных NULL или не содержащих ключей, согласно функции `extractValue`. Это позволяет находить при поиске пустые объекты, когда они должны быть найдены.

Составные индексы GIN реализуются в виде одного B-дерева по составным значениям (номер столбца, значение ключа). Значения ключей для различных столбцов могут быть разных типов.

60.4.1. Методика быстрого обновления GIN

Природа инвертированного индекса такова, что обновление GIN обычно медленная операция: при добавлении или изменении одной строки данных может потребоваться выполнить множество добавлений записей в индекс (для каждого ключа, извлечённого из индексируемого объекта). Начиная с PostgreSQL 8.4, GIN может отложить большой объём этой работы, вставляя новые кортежи во временный несортированный список записей, ожидающих индексации. Когда таблица очищается, автоматически анализируется, вызывается функция `gin_clean_pending_list` или размер этого временного списка становится больше чем gin_pending_list_limit, записи переносятся в основную структуру данных GIN теми же методами массового добавления данных, что и при начальном создании индекса. Это значительно увеличивает скорость обновления индекса GIN, даже с учётом дополнительных издержек при очистке. К тому же эту дополнительную работу можно выполнить в фоновом процессе, а не в процессе, непосредственно выполняющем запросы.

Основной недостаток такого подхода состоит в том, что при поиске необходимо не только проверить обычный индекс, но и просканировать список ожидающих записей, так что если этот список большой, поиск значительно замедляется. Ещё один недостаток состоит в том, что хотя в основном изменения производятся быстро, изменение, при котором этот список оказывается «слишком большим», влечёт необходимость немедленной очистки и поэтому выполняется гораздо дольше остальных изменений. Минимизировать эти недостатки можно, правильно организовав автоочистку.

Если выдержанность времени операций важнее скорости обновления, применение списка ожидающих записей можно отключить, выключив параметр хранения `fastupdate` для индекса GIN. За подробностями обратитесь к CREATE INDEX.

60.4.2. Алгоритм частичного соответствия

GIN может поддерживать проверки «частичного соответствия», когда запрос выявляет не точное соответствие одному или нескольким ключам, а возможные соответствия, попадающие в достаточно узкий диапазон значений ключей (при порядке сортировки ключей, определённым опорным методом `compare`). В этом случае метод `extractQuery` возвращает не значение ключа, которое должно соответствовать точно, а значение, определяющее нижнюю границу искомого диапазона, и устанавливает флаг `pmatch`. Затем диапазон ключей сканируется методом `comparePartial`. Метод `comparePartial` должен вернуть ноль при соответствии ключа индекса, отрицательное значение, если соответствия нет, но нужно продолжать проверку диапазона, и положительное значение, если ключ индекса оказался за искомым диапазоном.

60.5. Приёмы и советы по применению GIN

Создание или добавление

Добавление объектов в индекс GIN может выполняться медленно, так как для каждого объекта скорее всего потребуется добавлять множество ключей. Поэтому при массовом добавлении данных в таблицу рекомендуется удалить индекс GIN и пересоздать его по окончании добавления.

Начиная с PostgreSQL 8.4, этот совет менее актуален, так как выполнение индексации может быть отложенным (подробнее об этом в Подразделе 60.4.1). Но при очень большом объёме изменений может быть лучше всё-таки удалить и пересоздать индекс.

maintenance_work_mem

Время построения индекса GIN очень сильно зависит от параметра `maintenance_work_mem`; не стоит экономить на рабочей памяти при создании индекса.

gin_pending_list_limit

В процессе последовательных добавлений в существующий индекс GIN с включённым режимом `fastupdate`, система будет очищать список ожидающих индексации записей, когда его размер

будет превышать gin_pending_list_limit. Во избежание значительных колебаний конечного времени ответа имеет смысл проводить очистку этого списка в фоновом режиме (то есть, применяя автоочистку). Избежать операций очистки на переднем плане можно, увеличив gin_pending_list_limit или проводя автоочистку более активно. Однако если вследствие увеличения порога операции очистки запустится очистка на переднем плане, она будет выполняться ещё дольше.

Значение gin_pending_list_limit можно переопределить для отдельных индексов GIN, изменив их параметры хранения, что позволяет задавать для каждого индекса GIN свой порог очистки. Например, можно увеличить порог только для часто обновляемых индексов GIN и оставить его низким для остальных.

gin_fuzzy_search_limit

Основной целью разработки индексов GIN было обеспечить поддержку хорошо расширяемого полнотекстового поиска в Postgres Pro, а при полнотекстовом поиске нередко возникают ситуации, когда возвращается очень большой набор результатов. Однако чаще всего так происходит, когда запрос содержит очень часто встречающиеся слова, так что полученный результат всё равно оказывается бесполезным. Так как чтение множества записей с диска и последующая сортировка их может занять много времени, это неприемлемо в производственных условиях. (Заметьте, что поиск по индексу при этом выполняется очень быстро.)

Для управляемого выполнения таких запросов в GIN введено настраиваемое мягкое ограничение сверху для числа возвращаемых строк: конфигурационный параметр gin_fuzzy_search_limit. По умолчанию он равен 0 (то есть ограничение отсутствует). Если он имеет ненулевое значение, возвращаемый набор строк будет случайно выбранным подмножеством всего набора результатов.

«Мягким» оно называется потому, что фактическое число возвращаемых строк может несколько отличаться от заданного значения, в зависимости от запроса и качества системного генератора случайных чисел.

Из практики, со значениями в несколько тысяч (например, 5000 — 20000) получаются приемлемые результаты.

60.6. Ограничения

GIN полагает, что индексируемые операторы являются строгими. Это означает, что функция extractValue вовсе не будет вызываться для значений NULL (вместо этого будет автоматически создаваться пустая запись в индексе), так же как и extractQuery не будет вызываться с искомым значением NULL (при этом сразу предполагается, что запрос не удовлетворяется). Заметьте однако, что при этом поддерживаются ключи NULL, содержащиеся в составных объектах или искомых значениях.

60.7. Примеры

В базовый дистрибутив Postgres Pro включены классы операторов GIN, перечисленные ранее в Таблице 60.1. Также классы операторов GIN содержатся в следующих модулях contrib:

btree_gin

Функциональность B-дерева для различных типов данных

hstore

Модуль для хранения пар (ключ, значение)

intarray

Расширенная поддержка int[]

`pg_trgm`

Схожесть текста на основе статистики триграмм

Глава 61. Индексы BRIN

61.1. Введение

BRIN расшифровывается как «Block Range Index» (Индекс зон блоков). BRIN предназначается для обработки очень больших таблиц, в которых определённые столбцы некоторым естественным образом коррелируют с их физическим расположением в таблице. *Зоной блоков* называется группа страниц, физически расположенных в таблице рядом; для каждой зоны в индексе сохраняется некоторая сводная информация. Например, в таблице заказов магазина может содержаться поле с датой добавления заказа, и практически всегда записи более ранних заказов и в таблице будут размещены ближе к началу; в таблице, содержащей столбец с почтовым индексом, также естественным образом могут группироваться записи по городам.

Индексы BRIN могут удовлетворять запросы, выполняя обычное сканирование по битовой карте, и будут возвращать все кортежи во всех страницах каждой зоны, если сводные данные, сохранённые в индексе, *соответствуют* условиям запроса. Исполнитель запроса должен перепроверить эти кортежи и отбросить те, что не соответствуют условиям запроса — другими словами, эти индексы неточные. Так как индекс BRIN очень маленький, сканирование индекса влечёт мизерные издержки по сравнению с последовательным сканированием, но может избавить от необходимости сканирования больших областей таблицы, которые определённо не содержат подходящие кортежи.

Конкретные данные, которые будут хранится в индексе BRIN, а также запросы, которые сможет поддержать этот индекс, зависят от класса операторов, выбранного для каждого столбца индекса. Например, типы данных с линейным порядком сортировки могут иметь классы операторов, хранящие минимальное и максимальное значение для каждой зоны блоков; для геометрических типов может храниться прямоугольник, вмещающий все объекты в зоне блоков.

Размер зоны блоков определяется в момент создания индекса параметром хранения `pages_per_range`. Число записей в индексе будет равняться размеру отношения в страницах, делённому на установленное значение `pages_per_range`. Таким образом, чем меньше это число, тем больше становится индекс (так как в нём требуется хранить больше элементов), но в то же время сводные данные могут быть более точными и большее число блоков данных может быть пропущено при сканировании индекса.

61.1.1. Обслуживание индекса

Во время создания индекса сканируются все существующие страницы, и в результате в индексе создаётся сводный кортеж для каждой зоны, в том числе, возможно неполной зоны в конце. По мере того, как данными наполняются новые страницы, если они оказываются в зонах, для которых уже есть сводная информация, она будет обновлена с учётом данных из новых кортежей. Если же создаётся новая страница, которая не попадает в последнюю зону, для новой зоны автоматически не рассчитывается сводная запись; кортежи на таких страницах остаются неучтёнными, пока позже не будет проведён расчёт сводных данных. Эта процедура может быть вызвана вручную, с помощью функции `brin_summarize_new_values(regclass)`, или автоматически, когда таблицу будет обрабатывать `VACUUM` или при автоочистке по мере добавления записей. (Последний метод отключён по умолчанию и может быть включён параметром `autosummarize`.) И наоборот, можно удалить сводное значение для зоны, вызвав функцию `brin_desummarize_range(regclass, bigint)`, что может быть полезно, когда этот кортеж в индексе становится не очень хорошим представлением соответствующих данных, так как они изменились.

Когда включён режим автопересчёта сводки, при каждом заполнении зоны страниц механизму автоочистки передаётся запрос для пересчёта сводки только по этой зоне, и он будет выполнен в конце следующего прохода обработки той же базы данных. Если очередь запросов переполнена, запрос в неё не записывается и в журнал сервера выводится соответствующее сообщение:

```
LOG:  request for BRIN range summarization for index "brin_wi_idx" page 128 was not
 recorded
```

В этой ситуации сводка для данной зоны будет пересчитана при выполнении следующей обычной очистки таблицы.

61.2. Встроенные классы операторов

В базовый дистрибутив Postgres Pro включены классы операторов BRIN, перечисленные в Таблице 61.1.

Классы операторов *minmax* хранят минимальные и максимальные значения, встречающиеся в индексированном столбце в определённой зоне. Классы операторов *inclusion* хранят значение, в котором содержатся значения индексированного столбца в определённой зоне.

Таблица 61.1. Встроенные классы операторов BRIN

Имя	Индексируемый тип данных	Индексируемые операторы
abstime_minmax_ops	abstime	< <= = >= >
int8_minmax_ops	bigint	< <= = >= >
bit_minmax_ops	bit	< <= = >= >
varbit_minmax_ops	bit varying	< <= = >= >
box_inclusion_ops	box	<< &< && &> >> ~= @> <@ &<| <<| |>> |&>
bytea_minmax_ops	bytea	< <= = >= >
bpchar_minmax_ops	character	< <= = >= >
char_minmax_ops	"char"	< <= = >= >
date_minmax_ops	date	< <= = >= >
float8_minmax_ops	double precision	< <= = >= >
inet_minmax_ops	inet	< <= = >= >
network_inclusion_ops	inet	&& >>= <<= = >> <<
int4_minmax_ops	integer	< <= = >= >
interval_minmax_ops	interval	< <= = >= >
macaddr_minmax_ops	macaddr	< <= = >= >
macaddr8_minmax_ops	macaddr8	< <= = >= >
name_minmax_ops	name	< <= = >= >
numeric_minmax_ops	numeric	< <= = >= >
pg_lsn_minmax_ops	pg_lsn	< <= = >= >
oid_minmax_ops	oid	< <= = >= >
range_inclusion_ops	любой тип диапазона	<< &< && &> >> @> <@ -|- = < <= = > >=
float4_minmax_ops	real	< <= = >= >
reltime_minmax_ops	reltime	< <= = >= >
int2_minmax_ops	smallint	< <= = >= >
text_minmax_ops	text	< <= = >= >
tid_minmax_ops	tid	< <= = >= >
timestamp_minmax_ops	timestamp without time zone	< <= = >= >
timestamptz_minmax_ops	timestamp with time zone	< <= = >= >
time_minmax_ops	time without time zone	< <= = >= >
timetz_minmax_ops	time with time zone	< <= = >= >

Имя	Индексируемый тип данных	Индексируемые операторы
uuid_minmax_ops	uuid	< <= = >= >

61.3. Расширяемость

Интерфейс BRIN характеризуется высоким уровнем абстракции и таким образом требует от разработчика метода доступа реализовать только смысловое наполнение обрабатываемого типа данных. Уровень BRIN берёт на себя заботу о параллельном доступе, поддержке журнала и поиске в структуре индекса.

Всё, что нужно, чтобы получить работающий метод доступа BRIN — это реализовать несколько пользовательских методов, определяющих поведение сводных значений, хранящихся в индексе, и их взаимоотношения с ключами сканирования. Словом, BRIN сочетает расширяемость с универсальностью, повторным использованием кода и аккуратным интерфейсом.

Класс операторов для BRIN должен предоставлять четыре метода:

`BrinOpcInfo *opcInfo(Oid type_oid)`

Возвращает внутреннюю информацию о сводных данных индексированных столбцов. Возвращаемое значение должно указывать на `BrinOpcInfo` (в памяти palloc) со следующим определением:

```
typedef struct BrinOpcInfo
{
    /* Число полей, хранящихся в столбце индекса этого класса операторов */
    uint16      oi_nstored;

    /* Непрозрачный указатель для внутреннего использования классом операторов */
    void       *oi_opaque;

    /* Элементы кеша типов для сохранённых столбцов */
    TypeCacheEntry *oi_typcache[FLEXIBLE_ARRAY_MEMBER];
} BrinOpcInfo;
```

Поле `BrinOpcInfo.oi_opaque` могут использовать подпрограммы класса операторов для передачи информации опорным процедурам при сканировании индекса.

`bool consistent(BrinDesc *bdesc, BrinValues *column, ScanKey key)`

Показывает, соответствует ли значение ScanKey заданным индексированным значениям некоторой зоны. Номер целевого атрибута передаётся в составе ключа сканирования.

`bool addValue(BrinDesc *bdesc, BrinValues *column, Datum newval, bool isnull)`

Для заданного кортежа индекса и индексируемого значения изменяет выбранный атрибут кортежа, чтобы он дополнительно охватывал новое значение. Если в кортеж вносятся какие-либо изменения, возвращается `true`.

`bool unionTuples(BrinDesc *bdesc, BrinValues *a, BrinValues *b)`

Консолидирует два кортежа индекса. Получая два кортежа, изменяет выбранный атрибут первого из них, что он охватывал оба кортежа. Второй кортеж не изменяется.

Основной дистрибутив включает поддержку двух типов классов операторов: minmax и inclusion. Определения классов операторов, использующие их, представлены для встроенных типов данных, насколько это уместно. Пользователь может определить дополнительные классы операторов для других типов данных, применяя аналогичные определения, и обойтись таким образом без написания кода; достаточно будет объявить нужные записи в каталоге. Заметьте, что предположения о семантике стратегий операторов зашиты в исходном коде опорных процедур.

Также возможно создать классы операторов, воплощающие полностью другую семантику, разработав реализации четырёх основных опорных процедур, описанных выше. Заметьте, что

обратная совместимость между разными основными версиями не гарантируется: к примеру, в следующих выпусках могут потребоваться дополнительные опорные процедуры.

При написании класса операторов для типа данных, представляющего полностью упорядоченное множество, можно использовать опорные процедуры minmax вместе с соответствующими операторами, как показано в Таблице 61.2. Все члены класса операторов (процедуры и операторы) являются обязательными.

Таблица 61.2. Номера стратегий и опорных процедур для классов операторов minmax

Член класса операторов	Объект
Опорная процедура 1	внутренняя функция `brin_minmax_opcinfo()`
Опорная процедура 2	внутренняя функция `brin_minmax_add_value()`
Опорная процедура 3	внутренняя функция `brin_minmax_consistent()`
Опорная процедура 4	внутренняя функция `brin_minmax_union()`
Стратегия оператора 1	оператор меньше
Стратегия оператора 2	оператор меньше-или-равно
Стратегия оператора 3	оператор равно
Стратегия оператора 4	оператор больше-или-равно
Стратегия оператора 5	оператор больше

При написании класса операторов для сложного типа данных, значения которого включаются в другой тип, можно использовать опорные процедуры inclusion вместе с соответствующими операторами, как показано в Таблице 61.3. Для этого требуется одна дополнительная функция, которую можно написать на любом языке. Для расширенной функциональности можно определить другие функции. Все операторы являются необязательными. Некоторые из них требует наличия других, что показано в таблице как зависимости.

Таблица 61.3. Номера стратегий и опорных процедур для классов операторов inclusion

Член класса операторов	Объект	Зависимость
Опорная процедура 1	внутренняя функция `brin_inclusion_opcinfo()`	
Опорная процедура 2	внутренняя функция `brin_inclusion_add_value()`	
Опорная процедура 3	внутренняя функция `brin_inclusion_consistent()`	
Опорная процедура 4	внутренняя функция `brin_inclusion_union()`	
Опорная процедура 11	функция для слияния двух элементов	
Опорная процедура 12	необязательная функция для проверки возможности слияния двух элементов	
Опорная процедура 13	необязательная функция для проверки, содержится ли один элемент в другом	
Опорная процедура 14	необязательная функция для проверки, является ли элемент пустым	

Член класса операторов	Объект	Зависимость
Стратегия оператора 1	оператор левее	Стратегия оператора 4
Стратегия оператора 2	оператор не-простирается-правее	Стратегия оператора 5
Стратегия оператора 3	оператор перекрывается	
Стратегия оператора 4	оператор не-простирается-левее	Стратегия оператора 1
Стратегия оператора 5	оператор правее	Стратегия оператора 2
Стратегия оператора 6, 18	оператор то-же-или-равно	Стратегия оператора 7
Стратегия оператора 7, 13, 16, 24, 25	оператор содержит-или-равно	
Стратегия оператора 8, 14, 26, 27	оператор содержится-в-или-равно	Стратегия оператора 3
Стратегия оператора 9	оператор не-простирается-выше	Стратегия оператора 11
Стратегия оператора 10	оператор ниже	Стратегия оператора 12
Стратегия оператора 11	оператор выше	Стратегия оператора 9
Стратегия оператора 12	оператор не-простирается-ниже	Стратегия оператора 10
Стратегия оператора 20	оператор меньше	Стратегия оператора 5
Стратегия оператора 21	оператор меньше-или-равно	Стратегия оператора 5
Стратегия оператора 22	оператор больше	Стратегия оператора 1
Стратегия оператора 23	оператор больше-или-равно	Стратегия оператора 1

Номера опорных процедур 1-10 зарезервированы для внутренних функций BRIN, так что функции уровня SQL начинаются с номера 11. Опорная функция номер 11 является основной, необходимой для построения индекса. Она должна принимать два аргумента того же типа данных, что и целевой тип класса, и возвращать их объединение. Класс операторов inclusion может сохранять значения объединения в различных типах данных, в зависимости от параметра `STORAGE`. Возвращаемое функцией объединения значение должно соответствовать типу данных `STORAGE`.

Опорные процедуры под номерами 12 и 14 предоставляются для поддержки нерегулярностей встроенных типов данных. Процедура номер 12 применяется для поддержки работы с сетевыми адресами из различных семейств, которые нельзя объединять. Процедура номер 14 применяется для поддержки зон с пустыми значениями. Процедура номер 13 является необязательной, но рекомендуемой; она проверяет новое значение, прежде чем оно будет передано функции объединения. Так как инфраструктура BRIN может соптимизировать некоторые операции, когда объединение не меняется, то применяя эту функцию, можно увеличить быстродействие индекса.

Классы операторов minmax и inclusion поддерживают операторы с разными типами, хотя с ними зависимости становятся более сложными. Класс minmax требует, чтобы для двух аргументов одного типа определялся полный набор операторов. Это позволяет поддерживать дополнительные типы данных, определяя дополнительные наборы операторов. Стратегии операторов класса inclusion могут зависеть от других стратегий, как показано в Таблице 61.3, или от своих собственных стратегий. Для них требуется, чтобы был определён необходимый оператор с типом данных `STORAGE` для левого аргумента и другим поддерживаемым типом для правого аргумента реализуемого оператора. См. определение `float4_minmax_ops` в качестве примера для minmax и `box_inclusion_ops` в качестве примера для inclusion.

Глава 62. Хеш-индексы

62.1. Обзор

Postgres Pro поддерживает реализацию хеш-индексов, которые хранятся на диске и могут полностью восстанавливаться после сбоя. Хеш-индекс может быть построен по данным любого типа, в том числе типа, для которого не определён линейный порядок. Хеш-индексы хранят только хеш-значение индексируемых данных, поэтому размер индексируемого столбца данных неограничен.

Хеш-индексы могут строиться только по одному столбцу и не позволяют проверять уникальность.

Хеш-индексы поддерживают только оператор =, поэтому для предложений WHERE, в которых фигурируют проверки интервалов, хеш-индексы будут бесполезны.

Каждый кортеж хеш-индекса хранит только 4-байтовое хеш-значение, а не фактическое значение столбца. В результате хеш-индексы могут быть намного меньше, чем B-деревья, при индексировании более длинных элементов данных, таких как UUID, URL-адреса и т. д. Поскольку значения столбцов в хеш-индексе отсутствуют, при его сканировании результаты будут неточными. Хеш-индексы могут участвовать в сканировании индекса по битовой карте и обратном сканировании.

Хеш-индексы предназначены в первую очередь для нагрузки с большим количеством операций SELECT и UPDATE, которые выполняют сканирование с проверкой равенства для больших таблиц. В индексе-B-дереве поиск должен проходить по дереву до тех пор, пока не будет найдена листовая страница. В таблицах с миллионами строк такой «спуск» может увеличить время доступа к данным. Листовым страницам в хеш-индексе соответствуют страницы ячеек. В отличие от индекса-B-дерева хеш-индекс позволяет напрямую обращаться к страницам ячеек, тем самым потенциально сокращая время доступа к индексу в больших таблицах. Это сокращение «логического ввода-вывода» становится ещё более заметным для индексов/данных, которые не умещаются в общих буферах/ОЗУ.

Конструкция хеш-индексов в принципе приспособлена для индексирования данных с неравномерным распределением хешей, однако наиболее эффективен в них прямой доступ к страницам ячеек, осуществляемый при равномерном распределении значений хеша. Когда при добавлении нового значения оказывается, что страница ячеек заполнена, к этой странице привязываются дополнительные страницы переполнения, локально расширяющие хранилище для индексных кортежей, соответствующих этому значению хеша. Затем в ходе выполнения последующих запросов при сканировании ячейки хеша потребуется просканировать все страницы переполнения. Таким образом, несбалансированный хеш-индекс для некоторых данных фактически может оказаться хуже индекса-B-дерева по количеству требуемых обращений к блокам.

Ввиду неэффективности хеш-индексов в случаях переполнения можно сказать, что они лучше всего подходят для уникальных данных, данных с низкой степенью уникальности или данных с небольшим количеством строк в одной ячейке хеша. Один из возможных способов избежать проблем — исключить из индекса значения с низкой степенью уникальности, определив частичный индекс по условию, но такой вариант может подойти далеко не всегда.

Как и B-деревья, хеш-индексы выполняют простое удаление индексных кортежей. Это отложенная операция обслуживания, удаляющая индексные кортежи, о которых известно, что их можно безопасно стереть (то есть те, для идентификаторов которых установлен бит LP_DEAD). Если при добавлении нового кортежа обнаруживается, что на странице нет свободного места, данная операция пытается предотвратить создание новой страницы переполнения, удаляя мёртвые индексные кортежи (что возможно, только если страница не закреплена). Удаление мёртвых индексных указателей также происходит во время операции VACUUM.

Если возможно, операция VACUUM также попытается уместить индексные кортежи в наименьшем количестве страниц переполнения, минимизируя цепочку переполнения. Если страницы

переполнения становятся пустыми, они могут быть переработаны для повторного использования другими ячейками, но они никогда не возвращаются операционной системе. В настоящее время единственная возможность уменьшить размер хеш-индекса — перестроить его командой REINDEX. Также на данный момент невозможно уменьшить количество ячеек.

Хеш-индексы могут расширяться за счёт увеличения количества страниц ячеек по мере увеличения числа индексируемых строк. Сопоставление значения хеша с номером ячейки выбирается таким образом, чтобы индекс мог расширяться последовательно. Когда в индекс нужно будет добавить новую ячейку, «разделить» придётся ровно одну существующую ячейку, при этом некоторые из её кортежей будут перенесены в новую ячейку в соответствии с изменённым сопоставлением.

Это расширение выполняется на переднем плане, и из-за него может увеличиваться время добавления данных для пользователей. Поэтому хеш-индексы могут не подходить для таблиц с быстро увеличивающимся числом строк.

62.2. Реализация

В хеш-индексе есть четыре типа страниц: метастраница (нулевая страница), содержащая статически размещённую управляющую информацию, основные страницы ячеек, страницы переполнения и страницы битовой карты, в которых отслеживаются освободившиеся страницы переполнения, пригодные для повторного использования. Страницы битовой карты для целей адресации рассматриваются как подмножество страниц переполнения.

Как сканирование индекса, так и добавление кортежей требует вычисления ячейки, в которой должен располагаться данный кортеж. Для этого нужно знать количество ячеек и значения `highmask` и `lowmask` из метастраницы; однако для оптимальной производительности нежелательно блокировать и закреплять метастраницу при каждой такой операции. Поэтому в кеше отношений каждого обслуживающего процесса имеется кешированная копия метастраницы. Благодаря этому обеспечивается правильное сопоставление ячеек при условии, что целевая ячейка не была разделена с момента последнего обновления кеша.

Основные страницы ячеек и страницы переполнения выделяются независимо, поскольку для каждого конкретного индекса может потребоваться больше или меньше страниц переполнения относительно количества ячеек в нём. В расчёте хеш-кодов используется интересный набор правил, позволяющий поддерживать переменное количество страниц переполнения, не требуя перемещения основных страниц ячеек после их создания.

Каждая проиндексированная строка в таблице представлена в хеш-индексе одним индексным кортежем. Кортежи хеш-индекса хранятся в страницах ячеек и страницах переполнения (если они есть). Для ускорения поиска элементы индекса размещаются в каждой конкретной странице упорядоченными по хеш-коду, что позволяет использовать в рамках страницы двоичный поиск. Однако заметьте, что никаких предположений об относительном порядке хеш-кодов в разных страницах, относящихся к одной ячейки, делать нельзя.

Алгоритмы разделения ячеек для расширения хеш-индекса слишком сложны, чтобы упоминать о них здесь. Алгоритм разделения защищён от сбоев и может быть перезапущен, если разделение не было успешно завершено.

Глава 63. Физическое хранение базы данных

В данной главе рассматривается формат физического хранения, используемый базами данных Postgres Pro.

63.1. Размещение файлов базы данных

В данном разделе описывается формат хранения на уровне файлов и каталогов.

Файлы конфигурации и файлы данных, используемые кластером базы данных, традиционно хранятся вместе в каталоге данных, который обычно называют `PGDATA` (по имени переменной среды, которую можно использовать для его определения). Обычно `PGDATA` находится в `/var/lib/pgpro/std-10/data`. На одной и той же машине может находиться множество кластеров, управляемых различными экземплярами сервера.

В каталоге `PGDATA` содержится несколько подкаталогов и управляющих файлов, как показано в Таблице 63.1. В дополнение к этим обязательным элементам конфигурационные файлы кластера `postgresql.conf`, `pg_hba.conf` и `pg_ident.conf` традиционно хранятся в `PGDATA`, хотя их можно разместить и в другом месте.

Таблица 63.1. Содержание `PGDATA`

Элемент	Описание
`PG_VERSION`	Файл, содержащий номер основной версии Postgres Pro
`base`	Подкаталог, содержащий подкаталоги для каждой базы данных
`current_logfiles`	Файл, в котором отмечается, в какие файлы журналов производит запись сборщик сообщений
`global`	Подкаталог, содержащий общие таблицы кластера, такие как `pg_database`
`pg_commit_ts`	Подкаталог, содержащий данные о времени фиксации транзакций
`pg_dynshmem`	Подкаталог, содержащий файлы, используемые подсистемой динамически разделяемой памяти
`pg_logical`	Подкаталог, содержащий данные о состоянии для логического декодирования
`pg_multixact`	Подкаталог, содержащий данные о состоянии мультитранзакций (используемые для разделяемой блокировки строк)
`pg_notify`	Подкаталог, содержащий данные состояния прослушивания и уведомлений (LISTEN/NOTIFY)
`pg_replslot`	Подкаталог, содержащий данные слота репликации
`pg_serial`	Подкаталог, содержащий информацию о выполненных сериализуемых транзакциях.
`pg_snapshots`	Подкаталог, содержащий экспортированные снимки (snapshots)
`pg_stat`	Подкаталог, содержащий постоянные файлы для подсистемы статистики.

Элемент	Описание
pg_stat_tmp	Подкаталог, содержащий временные файлы для подсистемы статистики
pg_subtrans	Подкаталог, содержащий данные о состоянии подтранзакций
pg_tblspc	Подкаталог, содержащий символические ссылки на табличные пространства
pg_twophase	Подкаталог, содержащий файлы состояний для подготовленных транзакций
pg_wal	Подкаталог, содержащий файлы WAL (журнал предзаписи)
pg_xact	Подкаталог, содержащий данные о состоянии транзакции
postgresql.auto.conf	Файл, используемый для хранения параметров конфигурации, которые устанавливаются при помощи ALTER SYSTEM
postmaster.opts	Файл, содержащий параметры командной строки, с которыми сервер был запущен в последний раз
postmaster.pid	Файл блокировки, содержащий идентификатор (ID) текущего управляющего процесса (PID), путь к каталогу данных кластера, время запуска управляющего процесса, номер порта, путь к каталогу Unix-сокета (пустой для Windows), первый корректный адрес прослушивания (listen_address) (IP-адрес или *, либо пустое значение в случае отсутствия прослушивания по TCP), и ID сегмента разделяемой памяти (этот файл отсутствует после остановки сервера).

Для каждой базы данных в кластере существует подкаталог внутри PGDATA/base, названный по OID базы данных в pg_database. Этот подкаталог по умолчанию является местом хранения файлов базы данных; в частности, там хранятся её системные каталоги.

Каждая таблица и индекс хранятся в отдельном файле. Для обычных отношений, эти файлы получают имя по номеру *файлового узла* таблицы или индекса, который содержится в pg_class.relfilenode. Но для временных отношений, имя файла имеет форму t*BBB*_*FFF*, где *BBB* - идентификатор серверного процесса сервера, который создал данный файл, а *FFF* — номер файлового узла. В обоих случаях, помимо главного файла (также называемого основным слоем), у каждой таблицы и индекса есть *карта свободного пространства* (см. Раздел 63.3), в которой хранится информация о свободном пространстве в данном отношении. Имя файла карты свободного пространства образуется из номера файлового узла с суффиксом _fsm. Также таблицы имеют *карту видимости*, хранящуюся в слое с суффиксом _vm, для отслеживания страниц, не содержащих мёртвых записей. Карта видимости подробнее описана в Разделе 63.4. Нежурналируемые таблицы и индексы имеют третий слой, так называемый слой инициализации, имя которого содержит суффикс _init (см. Раздел 63.5).

Внимание

Заметьте, что хотя номер файла таблицы часто совпадает с её OID, так бывает *не* всегда; некоторые операции, например, TRUNCATE, REINDEX, CLUSTER и некоторые формы команды ALTER TABLE могут изменить номер файла, но при этом сохранят OID. Не следует рассчитывать, что номер файлового узла и OID таблицы совпадают. Кроме того, для некоторых системных каталогов, включая и pg_class, в pg_class.relfilenode содержится

ноль. Фактический номер файлового узла для них хранится в низкоуровневой структуре данных, и его можно получить при помощи функции `pg_relation_filenode()`.

Когда объём таблицы или индекса превышает 1 GB, они делятся на *сегменты* размером в один гигабайт. Файл первого сегмента называется по номеру файлового узла (filenode); последующие сегменты получают имена filenode.1, filenode.2 и т. д. При такой организации хранения не возникает проблем на платформах, имеющих ограничения по размеру файлов. (На самом деле, 1 ГБ — лишь размер по умолчанию. Размер сегмента можно изменить при сборке Postgres Pro, используя параметр конфигурации `--with-segsize`.) В принципе, карты свободного пространства и карты видимости также могут занимать нескольких сегментов, хотя на практике это маловероятно.

У таблицы, столбцы которой могут содержать данные большого объёма, будет иметься собственная таблица *TOAST*, предназначенная для отдельного хранения значений, которые слишком велики для хранения в строках самой таблицы. Основная таблица связывается с её таблицей TOAST (если таковая имеется) через `pg_class.reltoastrelid`. За подробной информацией обратитесь к Разделу 63.2.

Содержание таблиц и индексов рассматривается ниже (см. Раздел 63.6).

Табличное пространство делает сценарий более сложным. Каждое пользовательское табличное пространство имеет символическую ссылку внутри каталога `PGDATA/pg_tblspc`, указывающую на физический каталог табличного пространства (т. е., положение, указанное в команде табличного пространства `CREATE TABLESPACE`). Эта символическая ссылка получает имя по OID табличного пространства. Внутри физического каталога табличного пространства имеется подкаталог, имя которого зависит от версии сервера Postgres Pro, как например `PG_9.0_201008051`. (Этот подкаталог используется для того, чтобы последующие версии базы данных могли свободно использовать одно и то же местоположение, заданное в `CREATE TABLESPACE`.) Внутри каталога конкретной версии находится подкаталог для каждой базы данных, которая имеет элементы в табличном пространстве, названный по OID базы данных. Таблицы и индексы хранятся внутри этого каталога, используя схему именования файловых узлов. Табличное пространство `pg_default` недоступно через `pg_tblspc`, но соответствует `PGDATA/base`. Подобным же образом, табличное пространство `pg_global` недоступно через `pg_tblspc`, но соответствует `PGDATA/global`.

Функция `pg_relation_filepath()` показывает полный путь (относительно `PGDATA`) для любого отношения. Часто это избавляет от необходимости запоминать многие из приведённых выше правил. Но следует помнить, что эта функция выдаёт лишь имя первого сегмента основного слоя отношения, т. е. возможно, понадобится добавить номер сегмента и/или `_fsm`, `_vm` или `_init`, чтобы найти все файлы, связанные с отношением.

Временные файлы (для таких операций, как сортировка объёма данных большего, чем может уместиться в памяти) создаются внутри `PGDATA/base/pgsql_tmp` или внутри подкаталога `pgsql_tmp` каталога табличного пространства, если для них определено табличное пространство, отличное от `pg_default`. Имя временного файла имеет форму `pgsql_tmp`*`PPP`*`.`*`NNN`*, где *`PPP`* — PID серверного процесса, а *`NNN`* служит для разделения различных временных файлов этого серверного процесса.

63.2. TOAST

В данном разделе рассматривается TOAST (The Oversized-Attribute Storage Technique, Методика хранения сверхбольших атрибутов).

Postgres Pro использует фиксированный размер страницы (обычно 8 КБ), и не позволяет кортежам занимать несколько страниц. Поэтому непосредственно хранить очень большие значения полей невозможно. Для преодоления этого ограничения большие значения полей сжимаются и/или разбиваются на несколько физических строк. Это происходит незаметно для пользователя и на большую часть кода сервера влияет незначительно. Этот метод известен как TOAST (тост, или «лучшее после изобретения нарезанного хлеба»). Инфраструктура TOAST также применяется для оптимизации обработки больших значений данных в памяти.

Лишь определённые типы данных поддерживают TOAST — нет смысла производить дополнительные действия с типами данных, размер которых не может быть большим. Чтобы поддерживать TOAST, тип данных должен представлять значение переменной длины (*varlena*), в котором первое четырёхбайтовое слово любого хранящегося значения содержит общую длину значения в байтах (включая само это слово). Содержание оставшейся части значения TOAST не ограничивает. Специальные представления, в целом называемые *значениями в формате TOAST*, работают, манипулируя этим начальным словом длины и интерпретируя его по-своему. Таким образом, функции уровня C, работающие с типом данных, поддерживающим TOAST, должны аккуратно обращаться со входными значениями, которые могут быть в формате TOAST: входные данные могут и не содержать четырёхбайтовое слово длины и содержимое после него, пока не будут *распакованы*. (Обычно в таких ситуациях нужно использовать макрос `PG_DETOAST_DATUM` прежде чем что-либо делать с входным значением, но в некоторых случаях возможны и более эффективные подходы. За подробностями обратитесь к Подразделу 35.11.1.)

TOAST занимает два бита слова длины varlena (старшие биты на машинах с порядком байт от старшего к младшему, или младшие биты — при другом порядке байт), таким образом, логический размер любого значения в формате TOAST ограничивается 1 Гигабайтом (2^{30} - 1 байт). Когда оба бита равны нулю, значение является обычным, не в формате TOAST, и оставшиеся биты слова длины задают общий размер элемента данных (включая слово длины) в байтах. Когда установлен старший (или младший, в зависимости от архитектуры) бит, значение имеет однобайтовый заголовок вместо обычного четырёхбайтового, а оставшиеся биты этого байта задают общий размер элемента данных (включая байт длины) в байтах. Этот вариант позволяет экономно хранить значения короче 127 байт и при этом допускает расширение значения этого типа данных до 1 Гбайта при необходимости. Значения с однобайтовыми заголовками не выравниваются по какой-либо определённой границе, тогда как значения с четырёхбайтовыми заголовками выравниваются по границе минимум четырёх байт; это избавление от выравнивания даёт дополнительный выигрыш в объёме, очень ощутимый для коротких значений. В качестве особого случая, если все оставшиеся биты однобайтового заголовка равны нулю (что в принципе невозможно с учётом включения размера длины), значением является указатель на отдельно размещённые данные, с несколькими возможными вариантами, описанными ниже. Тип и размер такого *указателя TOAST* определяется кодом, хранящимся во втором байте значения. Наконец, когда старший (или младший, в зависимости от архитектуры) бит очищен, а соседний бит установлен, содержимое данных хранится в упакованном виде и должно быть распаковано перед использованием. В этом случае оставшиеся биты четырёхбайтового слова длины задают общий размер сжатых, а не исходных данных. Заметьте, что сжатие также возможно и для отделённых данных, но заголовок varlena не говорит, имеет ли оно место — это определяется содержимым, на которое указывает указатель TOAST.

Как уже было сказано, существуют разные варианты использования указателя TOAST. Самый старый и наиболее популярный вариант — когда он указывает на отделённые данные, размещённые в *TOAST-таблице*, которая отделена, но связана с таблицей, содержащей собственно указатель данных TOAST. Такой указатель на данные *на диске* создаётся кодом обработки TOAST, когда кортеж, сохраняемый на диск, оказывается слишком большим. Дополнительные подробности описаны в Подразделе 63.2.1. Кроме того, указатель TOAST может указывать на отделённые данные, размещённые где-то в памяти. Такие данные обязательно недолговременные и никогда не оказываются на диске, но этот механизм очень полезен для исключения копирования и избыточной обработки данных большого размера. Дополнительные подробности описаны в Подразделе 63.2.2.

В качестве метода сжатия внутренних и отделённых данных применяется довольно простой и очень быстрый представитель семейства алгоритмов LZ. Подробнее см. `src/common/pg_lzcompress.c`.

63.2.1. Отдельное размещение TOAST на диске

Если какие-либо столбцы таблицы хранятся в формате TOAST, у таблицы будет связанная с ней таблица TOAST, OID которой хранится в значении `pg_class.reltoastrelid` для данной таблицы. Размещаемые на диске TOAST-значения содержатся в таблице TOAST, что подробнее описано ниже.

Отделённые значения делятся на порции (после сжатия, если оно применяется) размером не более TOAST_MAX_CHUNK_SIZE байт (по умолчанию это значение выбирается таким образом, чтобы на странице помещались четыре строки порций, то есть размер одной составляет порядка 2000 байт). Каждая порция хранится как отдельная строка в таблице TOAST, принадлежащей исходной таблице-владельцу. Каждая таблица TOAST имеет столбцы chunk_id (OID, идентифицирующий конкретное TOAST-значение), chunk_seq (последовательный номер для порции внутри значения) и chunk_data (фактические данные порции). Уникальный индекс по chunk_id и chunk_seq обеспечивает быструю выдачу значений. Таким образом, в указателе, представляющем отдельно размещаемое на диске значение TOAST, должно храниться OID таблицы TOAST, к которой нужно обращаться, и OID определённого значения (его chunk_id). Для удобства в данных указателя также хранится логический размер элемента данных (исходных данных без сжатия) и фактический размер хранимых данных (отличающийся, если было применено сжатие). Учитывая байты заголовка varlena, общий размер указателя на хранимое на диске значение TOAST составляет 18 байт, независимо от фактического размера собственно значения.

Код обработки TOAST срабатывает, только когда значение строки, которое должно храниться в таблице, по размеру больше, чем TOAST_TUPLE_THRESHOLD байт (обычно это 2 Кб). Код TOAST будет сжимать и/или выносить значения поля за пределы таблицы до тех пор, пока значение строки не станет меньше TOAST_TUPLE_TARGET байт (также обычно 2 Кб) или уменьшить объём станет невозможно. Во время операции UPDATE значения неизменённых полей обычно сохраняются как есть, поэтому модификация строки с отдельно хранимыми значениями не несёт издержек, связанных с TOAST, если все такие значения остаются без изменений.

Код обработки TOAST распознаёт четыре различные стратегии хранения столбцов, совместимых с TOAST, на диске:

- PLAIN не допускает ни сжатия, ни отдельного хранения; кроме того, отключается использование однобайтовых заголовков для типов varlena. Это единственно возможная стратегия для столбцов типов данных, которые несовместимы с TOAST.
- EXTENDED допускает как сжатие, так и отдельное хранение. Это стандартный вариант для большинства типов данных, совместимых с TOAST. Сначала происходит попытка выполнить сжатие, затем — сохранение вне таблицы, если строка всё ещё слишком велика.
- EXTERNAL допускает отдельное хранение, но не сжатие. Использование EXTERNAL ускорит операции над частями строк в больших столбцах text и bytea (ценой увеличения объёма памяти для хранения), так как эти операции оптимизированы для извлечения только требуемых частей отделённого значения, когда оно не сжато.
- MAIN допускает сжатие, но не отдельное хранение. (Фактически для таких столбцов будет тем не менее применяться отдельное хранение, но лишь как крайняя мера, когда нет другого способа уменьшить строку так, чтобы она помещалась на странице.)

Каждый тип данных, совместимый с TOAST, определяет стандартную стратегию для столбцов этого типа данных, но стратегия для заданного столбца таблицы может быть изменена с помощью ALTER TABLE ... SET STORAGE.

Эта схема имеет ряд преимуществ по сравнению с более простым подходом, когда значения строк могут занимать несколько страниц. Если предположить, что обычно запросы характеризуются выполнением сравнения с относительно маленькими значениями ключа, большая часть работы будет выполняться с использованием главной записи строки. Большие значения атрибутов в формате TOAST будут просто передаваться (если будут выбраны) в тот момент, когда результирующий набор отправляется клиенту. Таким образом, главная таблица получается гораздо меньше, и в общий кеш буферов помещается больше её строк, чем их было бы без использования отдельного хранения. Наборы данных для сортировок также уменьшаются, а сортировки чаще будут выполняться исключительно в памяти. Небольшой тест показал, что таблица, содержащая типичные HTML-страницы и их URL после сжатия занимала примерно половину объёма исходных данных, включая таблицу TOAST, и что главная таблица содержала лишь около 10% всех данных (URL и некоторые маленькие HTML-страницы). Время обработки не отличалось от времени, необходимого для обработки таблицы без использования TOAST, в которой размер всех HTML-страниц был уменьшен до 7 Кб, чтобы они уместились в строках.

63.2.2. Отдельное размещение TOAST в памяти

Указатели TOAST могут указывать на данные, размещённые не на диске, а где-либо в памяти текущего серверного процесса. Очевидно, что такие указатели не могут быть долговременными, но они тем не менее полезны. В настоящее время поддерживаются два подварианта: *косвенные* указатели на данные и указатели на *развёрнутые* данные.

Косвенный указатель TOAST просто указывает на значение varlena, хранящееся где-то в памяти. Этот вариант изначально был реализован просто как подтверждение концепции, но в настоящее время он применяется при логическом декодировании, чтобы не приходилось создавать физические кортежи больше одного 1 ГБ (что может потребоваться при консолидации всех отделённых значений полей в одном кортеже). Данный вариант имеет ограниченное применение, так как создатель такого указателя должен полностью понимать, что целевые данные будут существовать, только пока существует указатель, и никакой инфраструктуры для сохранения их нет.

Указатели на развёрнутые данные TOAST полезны для сложных типов, представление которых на диске плохо приспособлено для вычислительных целей. Например, стандартное представление в виде varlena массива Postgres Pro включает информацию о размерности, битовую карту элементов NULL (если они в нём содержатся), а затем значения всех элементов по порядку. Когда элемент сам по себе имеет переменную длину, единственный способ найти *N*-ый элемент — просканировать все предыдущие элементы. Это представление компактно и поэтому подходит для хранения на диске, но для вычислительной обработки массива гораздо удобнее иметь «развёрнутое» или «деконструированное» представление, в котором можно определить начальные адреса всех элементов. Механизм указателей TOAST способствует решению этой задачи, допуская передачу по ссылке элемента Datum как указателя на стандартное значение varlena (представление на диске) или указателя TOAST на развёрнутое представление где-то в памяти. Детали развёрнутого представление определяются самим типом данных, хотя оно может иметь стандартный заголовок и удовлетворять другим требованиям API, описанным в `src/include/utils/expandeddatum.h`. Функции уровня C, работающие с этим типом, могут реализовать поддержку любого из этих представлений. Функции, не знающие о развёрнутом представлении, а просто применяющие `PG_DETOAST_DATUM` к своим входным данным, будут автоматически получать традиционное представление varlena; так что поддержка развёрнутого представления может вводиться постепенно, по одной функции.

Указатели TOAST на развёрнутые значения далее подразделяются на указатели *для чтения/записи* и указатели *только для чтения*. Представление, на которое они указывают, в любом случае одинаковое, но функции, получающей указатель для чтения/записи, разрешается модифицировать целевые данные прямо на месте, тогда как функция, получающая указатель только для чтения, не должна этого делать; если ей нужно получить изменённую версию значения, она должна сначала сделать копию. Это отличие и связанные с ним соглашения позволяют избежать излишнего копирования развёрнутых значений при выполнении запросов.

Для всех типов указателей TOAST на данные в памяти, код обработки TOAST гарантирует, что такие данные не окажутся случайно сохранены на диске. Указатели TOAST в памяти автоматически сворачиваются в обычные значения varlena перед сохранением — а затем могут преобразоваться в указатели TOAST на диске, если без этого не смогут уместиться в содержащем их кортеже.

63.3. Карта свободного пространства

Каждое табличное и индексное отношение, за исключением хеш-индексов, имеет карту свободного пространства (Free Space Map, FSM) для отслеживания доступного места. Она хранится рядом с данными главного отношения в отдельном слое, имя которого образуется номером файлового узла отношения с суффиксом `_fsm`. Например, если файловый узел отношения — 12345, FSM хранится в файле с именем `12345_fsm` в том же каталоге, что и основной файл отношения.

Карта свободного пространства представляет собой дерево страниц FSM. Страницы FSM нижнего уровня хранят информацию о свободном пространстве, доступном на каждой странице таблицы

(или индекса), используя один байт для представления каждой такой страницы. Верхние уровни агрегируют информацию нижних уровней.

Внутри каждой страницы FSM имеется двоичное дерево, хранящееся в массиве, где один байт выделяется на каждый узел дерева. Каждый листовой узел представляет страницу таблицы или страницу FSM нижнего уровня. В каждом узле выше листовых хранится наибольшее из значений его узлов-потомков. Поэтому максимальное из значений листовых узлов хранится в корневом узле.

Для просмотра информации, хранящейся в картах свободного пространства, можно воспользоваться модулем pg_freespacemap.

63.4. Карта видимости

Каждое отношение таблицы имеет карту видимости (Visibility Map, VM) для отслеживания страниц, содержащих только кортежи, которые видны всем активным транзакциям; в ней также отслеживается, какие страницы содержат только замороженные кортежи. Она хранится вместе с данными главного отношения в отдельном файле, имя которого образуется номером файлового узла отношения с суффиксом `_vm`. Например, если файловый узел отношения — 12345, VM хранится в файле `12345_vm`, в том же самом каталоге, что и основной файл отношения. Заметьте, что индексы не имеют VM.

Карта видимости хранит по два бита на страницу таблицы. Первый бит, если он установлен, показывает, что вся страница видна или, другими словами, не содержит кортежей, которые необходимо очистить. Эта информация может также использоваться при *сканировании только индекса* для поиска ответов только в данных индекса. Установленный второй бит показывает, что все кортежи на этой странице заморожены. Это означает, что процесс очистки для предотвращения зацикливания не должен больше посещать эту страницу.

Карта может отражать реальные данные с запаздыванием в том смысле, что мы уверены, что в случаях, когда установлен бит, известно, что условие верно, но если бит не установлен, оно может быть верным или неверным. Биты карты видимости устанавливаются только при очистке, а сбрасываются при любых операциях, изменяющих данные на странице.

Для изучения информации, хранящейся в карте видимости, можно воспользоваться модулем pg_visibility.

63.5. Слой инициализации

Каждая нежурналируемая таблица, и каждый индекс такой таблицы имеет файл инициализации. Файл инициализации представляет собой пустую таблицу или индекс соответствующего типа. Когда нежурналируемая таблица должна быть заново очищена по причине сбоя, файл инициализации копируется поверх главного файла, а все прочие файлы удаляются (при необходимости они будут автоматически созданы заново).

63.6. Компоновка страницы базы данных

В данном разделе рассматривается формат страницы, используемый в таблицах и индексах Postgres Pro.[1] Последовательности и таблицы TOAST форматируются как обычные таблицы.

В дальнейшем подразумевается, что *байт* содержит 8 бит. В дополнение, термин *элемент* относится к индивидуальному значению данных, которое хранится на странице. В таблице элемент — это строка; в индексе — элемент индекса.

Каждая таблица и индекс хранятся как массив *страниц* фиксированного размера (обычно 8 kB, хотя можно выбрать другой размер страницы при компиляции сервера). В таблице все страницы логически эквивалентны, поэтому конкретный элемент (строка) может храниться на любой странице. В индексах первая страница обычно резервируется как *метастраница*, хранящая

[1] Фактически индексные методы доступа не нуждаются в этом формате страниц. Все существующие индексные методы в действительности используют этот основной формат, но данные, хранящиеся в индексных метастраницах обычно не следуют правилам компоновки.

контрольную информацию, а внутри индекса могут быть разные типы страниц, в зависимости от метода доступа индекса.

Таблица 63.2 показывает общую компоновку страницы. Каждая страница имеет пять частей.

Таблица 63.2. Общая компоновка страницы

Элемент	Описание
Данные заголовка страницы	Длина — 24 байта. Содержит общую информацию о странице, включая указатели свободного пространства.
Данные идентификаторов элементов	Массив идентификаторов, указывающих на фактические элементы. Каждый идентификатор представляет собой пару «смещение, длина» и занимает 4 байта.
Свободное пространство	Незанятое пространство. Новые идентификаторы элементов размещаются с начала этой области, сами новые элементы — с конца.
Элементы	Сами элементы данных как таковые.
Специальное пространство	Специфические данные метода доступа. Для различных методов хранятся различные данные. Для обычных таблиц таких данных нет.

Первые 24 байта каждой страницы образуют заголовок страницы (`PageHeaderData`). Его формат подробно описан в Таблице 63.3. В первом поле отслеживается самая последняя запись в WAL, связанная с этой страницей. Второе поле содержит контрольную сумму страницы, если включён режим Контрольные суммы данных. Затем идёт двухбайтовое поле, содержащее биты флагов. За ним следуют три двухбайтовых целочисленных поля (`pd_lower`, `pd_upper` и `pd_special`). Они содержат смещения в байтах от начала страницы до начала незанятого пространства, до конца незанятого пространства и до начала специального пространства. В следующих 2 байтах заголовка страницы, в поле `pd_pagesize_version`, хранится размер страницы и индикатор версии. Начиная с PostgreSQL 8.3, используется версия 4; в PostgreSQL 8.1 и 8.2 использовалась версия 3; в PostgreSQL 8.0 — версия 2; в PostgreSQL 7.3 и 7.4 — версия 1; в предыдущих выпусках — версия 0. (Основная структура страницы и формат заголовка почти во всех этих версиях одни и те же, но структура заголовка строк в куче изменялась.) Размер страницы присутствует в основном только для перекрёстной проверки; возможность использовать в одной инсталляции разные размеры страниц не поддерживается. Последнее поле подсказывает, насколько вероятна возможность получить выигрыш, произведя очистку страницы: оно отслеживает самый старый XMAX на странице, не подвергавшийся очистке.

Таблица 63.3. Данные заголовка страницы (PageHeaderData)

Поле	Тип	Длина	Описание
pd_lsn	PageXLogRecPtr	8 байт	LSN: Следующий байт после последнего байта записи WAL для последнего изменения на этой странице
pd_checksum	uint16	2 байта	Контрольная сумма страницы
pd_flags	uint16	2 байта	Биты признаков
pd_lower	LocationIndex	2 байта	Смещение до начала свободного пространства

Поле	Тип	Длина	Описание
pd_upper	LocationIndex	2 байта	Смещение до конца свободного пространства
pd_special	LocationIndex	2 байта	Смещение до начала специального пространства
pd_pagesize_version	uint16	2 байта	Информация о размере страницы и номере версии компоновки
pd_prune_xid	TransactionId	4 байта	Самый старый неочищенный идентификатор XMAX на странице или ноль при отсутствии такового

Всю подробную информацию можно найти в `src/include/storage/bufpage.h`.

За заголовком страницы следуют идентификаторы элемента (`ItemIdData`), каждому из которых требуется 4 байта. Идентификатор элемента содержит байтовое смещение до начала элемента, его длину в байтах и несколько битов атрибутов, которые влияют на его интерпретацию. Новые идентификаторы элементов размещаются по мере необходимости от начала свободного пространства. Количество имеющихся идентификаторов элементов можно определить через значение `pd_lower`, которое увеличивается при добавлении нового идентификатора. Поскольку идентификатор элемента никогда не перемещается до тех пор, пока он не освобождается, его индекс можно использовать в течение длительного периода времени, чтобы ссылаться на элемент, даже когда сам элемент перемещается по странице для уплотнения свободного пространства. Фактически каждый указатель на элемент (`ItemPointer`, также известный как `CTID`), созданный Postgres Pro, состоит из номера страницы и индекса идентификатора элемента.

Сами элементы хранятся в пространстве, выделяемом в направлении от конца к началу незанятого пространства. Точная структура меняется в зависимости от того, каким будет содержимое таблицы. Как таблицы, так и последовательности используют структуру под названием `HeapTupleHeaderData`, которая описывается ниже.

Последний раздел является «особым разделом», который может содержать всё, что необходимо методу доступа для хранения. Например, индексы-B-деревья хранят ссылки на страницы слева и справа, равно как и некоторые другие данные, соответствующие структуре индекса. Обычные таблицы не используют особый раздел вовсе (что указывается установкой значения `pd_special` равным размеру страницы).

Все строки таблицы имеют одинаковую структуру. Они включают заголовок фиксированного размера (занимающий 23 байта на большинстве машин), за которым следует необязательная битовая карта пустых значений, необязательное поле идентификатора объекта и данные пользователя. Подробное описание заголовка представлено в Таблице 63.4. Актуальные пользовательские данные (столбцы строки) начинаются после смещения, заданного в `t_hoff`, которое должно всегда быть кратным величине MAXALIGN для платформы. Битовая карта пустых значений имеется тогда, когда бит *HEAP_HASNULL* установлен в значении `t_infomask`. В случае наличия, она начинается сразу после фиксированного заголовка и занимает достаточно байтов, чтобы иметь один бит на столбец (т. е. `t_natts` битов всего). В этом списке битов установленный в единицу бит означает непустое значение, а установленный в ноль соответствует пустому значению. Когда битовая карта отсутствует, все столбцы считаются непустыми. Идентификатор объекта присутствует, если только бит *HEAP_HASOID* установлен в значении `t_infomask`. Если он есть, он расположен сразу перед началом `t_hoff`. Любое заполнение, необходимое для того, чтобы сделать `t_hoff` кратным MAXALIGN, будет расположено между битовой картой пустых значений

и идентификатором объекта. (Это в свою очередь гарантирует, что идентификатор объекта будет правильно выровнен.)

Таблица 63.4. Данные заголовка строки таблицы (HeapTupleHeaderData)

Поле	Тип	Длина	Описание
t_xmin	TransactionId	4 байта	значение XID вставки
t_xmax	TransactionId	4 байта	значение XID удаления
t_cid	CommandId	4 байта	значение CID для вставки и/или удаления (пересекается с t_xvac)
t_xvac	TransactionId	4 байта	XID для операции VACUUM, которая перемещает версию строки
t_ctid	ItemPointerData	6 байт	текущее значение TID этой или более новой версии строки
t_infomask2	uint16	2 байта	количество атрибутов плюс различные биты флагов
t_infomask	uint16	2 байта	различные биты флагов
t_hoff	uint8	1 байт	отступ до пользовательских данных

Всю подробную информацию можно найти в `src/include/access/htup_details.h`.

Интерпретировать текущие данные можно только с использованием информации, полученной из других таблиц, в основном из `pg_attribute`. Ключевыми значениями, необходимыми для определения расположения полей, являются `attlen` и `attalign`. Не существует способа непосредственного получения заданного атрибута, кроме случая, когда имеются только поля фиксированной длины и при этом нет значений NULL. Все эти особенности учитываются в функциях *heap_getattr*, *fastgetattr* и *heap_getsysattr*.

Чтобы прочитать данные, необходимо просмотреть каждый атрибут по очереди. В первую очередь нужно проверить, является ли значение поля пустым согласно битовой карте пустых значений. Если это так, можно переходить к следующему полю. Затем следует убедиться, что выравнивание является верным. Если это поле фиксированной ширины, берутся просто все его байты. Если это поле переменной длины (attlen = -1), всё несколько сложнее. Все типы данных с переменной длиной имеют общую структуру заголовка `struct varlena`, которая включает общую длину сохранённого значения и некоторые биты флагов. В зависимости от установленных флагов, данные могут храниться либо локально, либо в таблице TOAST. Также, возможно сжатие данных (см. Раздел 63.2).

Глава 64. Внутренний интерфейс BKI

Файлы внутреннего интерфейса (BKI, Backend Interface) представляют собой скрипты на специальном языке, который понимает сервер Postgres Pro в режиме «первого запуска». Этот режим позволяет создать системные каталоги и заполнить их с нуля, тогда как для обычных SQL-команд необходимо, чтобы каталоги уже существовали. Таким образом, файлы BKI могут применяться для изначального создания системы баз данных. (И вряд ли им можно найти другое применение.)

Программа initdb использует файл BKI для выполнения части своей работы при создании нового кластера баз данных. Входной файл для initdb создаётся в процессе сборки и установки Postgres Pro программой `genbki.pl`, которая считывает для этого специально отформатированные заголовочные файлы C в каталоге `src/include/catalog/` в дереве исходного кода. Созданный файл BKI называется `postgres.bki` и обычно устанавливается в подкаталог `share` дерева инсталляции.

Дополнительные сведения можно найти в документации по initdb.

64.1. Формат файла BKI

В этом разделе описывается, как сервер Postgres Pro интерпретирует файлы BKI. Это описание будет легче понять, если для наглядности вы откроете файл `postgres.bki`.

Содержимое BKI состоит из последовательности команд. Команды образуются из нескольких компонентов, в зависимости от синтаксиса конкретной команды. Компоненты команд обычно разделяются пробельными символами, но это не обязательно, если не возникает неоднозначности. Специальный разделитель команд отсутствует; следующий компонент, который не может синтаксически относиться к предыдущей команде, начинает следующую. (Обычно новая команда начинается в отдельной строке, для структурности.) Компонентами команд могут быть определённые ключевые слова, специальные символы (скобки, запятые и т. д.), числа или строки в двойных кавычках. Все буквы в них воспринимаются с учётом регистра.

Строки, начинающиеся с `#`, игнорируются.

64.2. Команды BKI

`create` *`имя_таблицы`* *`oid_таблицы`* [`bootstrap`] [`shared_relation`] [`without_oids`] [`rowtype_oid` *`oid`*] (*`имя1`* = *`тип1`* [FORCE NOT NULL | FORCE NULL] [, *`имя2`* = *`тип2`* [FORCE NOT NULL | FORCE NULL], ...])

Создать таблицу *`имя_таблицы`* с заданным *`oid_таблицы`* и столбцами, указанными в скобках.

Непосредственно `bootstrap.c` поддерживает следующие типы столбцов: `bool`, `bytea`, `char` (1 байт), `name`, `int2`, `int4`, `regproc`, `regclass`, `regtype`, `text`, `oid`, `tid`, `xid`, `cid`, `int2vector`, `oidvector`, `_int4` (массив), `_text` (массив), `_oid` (массив), `_char` (массив), `_aclitem` (массив). Хотя возможно создать таблицы, содержащие столбцы и других типов, это нельзя сделать, пока не будет создан и заполнен соответствующими записями каталог `pg_type`. (Это по сути означает, что только эти типы столбцов могут быть в таблицах, создаваемых в таком режиме, хотя каталоги, создаваемые позже, могут содержать любые встроенные типы.)

С указанием `bootstrap` таблица будет создана только на диске; никакие записи о ней не будут добавлены в `pg_class`, `pg_attribute` и т. д. Таким образом, таблица не будет доступна для обычных операций SQL, пока такие записи не будут добавлены явно (командами `insert`). Это указание применяется для создания самой структуры `pg_class` и подобных ей.

Если добавлено указание `shared_relation`, таблица создаётся как общая. Она будет содержать столбец OID, если отсутствует указание `without_oids`. Дополнительным предложением `rowtype_oid` может быть задан OID типа строки (OID записи в `pg_type`); если он не указан, OID генерируется автоматически. (Предложение `rowtype_oid` бесполезно, если присутствует указание `bootstrap`, но его всё равно можно добавить для документирования.)

`open` *имя_таблицы*

Открыть таблицу *имя_таблицы* для добавления данных. Любая другая таблица, открытая в данный момент, закрывается.

`close` [*имя_таблицы*]

Закрыть открытую таблицу. Имя таблицы может задаваться для перепроверки, но это не требуется.

`insert` [`OID =` *значение_oid*] `(` *значение1 значение2* ... `)`

Вставить новую строку в открытую таблицу, установив *значение1*, *значение2* и т. д. в качестве значений столбцов и *значение_oid* в качестве OID. Если *значение_oid* равно нулю (0) или это указание опущено, а таблица при этом содержит OID, строке назначается следующий свободный OID.

Значения NULL могут задаваться специальным ключевым словом `_null_`. Значения, содержащие пробелы, должны заключаться в двойные кавычки.

`declare` [`unique`] `index` *имя_индекса oid_индекса* `on` *имя_таблицы* `using` *имя_метода_доступа* `(` *класс_оп1 имя1* [, ...] `)`

Создать индекс *имя_индекса* с OID, равным *oid_индекса*, в таблице *имя_таблицы*, с методом доступа *имя_метода_доступа*. Индекс строится по полям *имя1*, *имя2* и т. д., и для них используются соответственно классы операторов *класс_оп1*, *класс_оп2* и т. д. Эта команда создаёт файл индекса и добавляет соответствующие записи в каталог, но не инициализирует содержимое индекса.

`declare toast` *oid_таблицы_toast oid_индекса_toast* `on` *имя_таблицы*

Создаёт таблицу TOAST для таблицы *имя_таблицы*. Таблице TOAST назначается OID, равный *oid_таблицы_toast*, а её индексу назначается OID, равный *oid_индекса_toast*. Как и с `declare index`, заполнение индекса откладывается.

`build indices`

Заполнить индексы, объявленные ранее.

64.3. Структура файла BKI

Команда `open` может применяться, только когда открываемая ей таблица существует и для неё имеются записи в каталогах. (Минимальный набор этих каталогов образуют `pg_class`, `pg_attribute`, `pg_proc` и `pg_type`.) Чтобы можно было заполнить сами эти таблицы, команда `create` с указанием `bootstrap` неявно открывает создаваемую таблицу для добавления данных.

Кроме того, команды `declare index` и `declare toast` нельзя применять, пока не будут созданы и заполнены системные каталоги.

Таким образом, файл `postgres.bki` должен иметь следующую структуру:

1. `create bootstrap` (создание) одной из критичных таблиц
2. `insert` (добавление) данных, описывающих как минимум критичные таблицы
3. `close`
4. Повторение для других критичных таблиц.
5. `create` (создание) (без `bootstrap`) некритичной таблицы
6. `open`
7. `insert` (добавление) требуемых данных
8. `close`

9. Повторение для других некритичных таблиц.

10Определение индексов и таблиц TOAST.

11`build indices`

Несомненно есть и другие, недокументированные зависимости, диктующие определённый порядок.

64.4. Пример

Следующая последовательность команд создаст таблицу `test_table` с OID 420, имеющую два столбца `cola` и `colb` типа `int4` и `text`, соответственно, и вставит две строки в эту таблицу:

```
create test_table 420 (cola = int4, colb = text)
open test_table
insert OID=421 ( 1 "value1" )
insert OID=422 ( 2 _null_ )
close test_table
```

Глава 65. Как планировщик использует статистику

Данная глава основана на материалах, рассмотренных ранее (см. Раздел 14.1 и Раздел 14.2), и подробнее рассказывает о том, как планировщик использует статистику для определения количества строк, которое может вернуть каждая часть запроса. Это важная составляющая процесса создания плана запроса, предоставляющая большую часть исходного материала для расчёта стоимости.

Целью данной главы является не подробное документирование кода, а общее описание его работы. Возможно, это поможет тем, кто пожелает в дальнейшем ознакомиться с кодом.

65.1. Примеры оценки количества строк

В приведённых ниже примерах используются таблицы базы данных регрессионного тестирования Postgres Pro. Приведённые листинги получены в версии 8.3. Поведение более ранних (или поздних) версий может отличаться. Заметьте также, что поскольку команда `ANALYZE` использует случайную выборку при формировании статистики, после любого нового выполнения команды `ANALYZE` результаты незначительно изменятся.

Давайте начнём с очень простого запроса:

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

Как планировщик определяет мощность `tenk1`, рассматривается выше (см. Раздел 14.2), но для полноты здесь говорится об этом ещё раз. Количество страниц и строк берётся в `pg_class`:

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';

 relpages | reltuples
----------+-----------
      358 |     10000
```

Это текущие цифры, полученные при последнем выполнении команд `VACUUM` или `ANALYZE`, применённых к этой таблице. Затем планировщик выполняет выборку фактического текущего числа страниц в таблице (это недорогая операция, для которой не требуется сканирование таблицы). Если оно отличается от `relpages`, то `reltuples` изменяется для того, чтобы привести это значение к текущей оценке количества строк. В показанном выше примере значение `relpages` является актуальным, поэтому количество строк берётся равным `reltuples`.

Давайте обратимся к примеру с диапазонным условием в предложении `WHERE`:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;

                                 QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=24.06..394.64 rows=1007 width=244)
   Recheck Cond: (unique1 < 1000)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

Планировщик рассматривает условие предложения `WHERE` и находит в справочнике функцию избирательности для оператора `<` в `pg_operator`. Это значение содержится в столбце `oprrest`, и в данном случае значением является `scalarltsel`. Функция `scalarltsel` извлекает гистограмму для `unique1` из `pg_statistic`. Для вводимых вручную запросов удобнее просматривать более простое представление `pg_stats`:

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='unique1';

                    histogram_bounds
-----------------------------------------------------
 {0,993,1997,3050,4040,5036,5957,7057,8029,9016,9995}
```

Затем обрабатывается часть гистограммы, которая соответствует условию «< 1000». Таким образом и определяется избирательность. Гистограмма делит диапазон на равные частотные группы, поэтому нужно лишь определить группу, содержащую наше значение, и подсчитать её *долю* и *долю групп*, предшествующих данной. Очевидно, что значение 1000 находится во второй группе (993-1997). Если предположить, что внутри каждой группы распределение значений линейное, мы можем вычислить избирательность следующим образом:

```
selectivity = (1 + (1000 - bucket[2].min)/(bucket[2].max - bucket[2].min))/num_buckets
            = (1 + (1000 - 993)/(1997 - 993))/10
            = 0.100697
```

т. е. сумма элементов одной целой группы и пропорциональной части элементов второй, делённая на число групп. Теперь примерное число строк может быть рассчитано как произведение избирательности и мощности `tenk1`:

```
rows = rel_cardinality * selectivity
     = 10000 * 0.100697
     = 1007  (округлённо)
```

Далее, давайте рассмотрим пример с условием на равенство в предложении `WHERE`:

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'CRAAAA';

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=30 width=244)
   Filter: (stringu1 = 'CRAAAA'::name)
```

Планировщик вновь проверяет условие в предложении `WHERE` и определяет функцию избирательности для `=`, и этой функцией является `eqsel`. Для оценки равенства гистограмма бесполезна, вместо неё для оценки избирательности используется список *самых частых значений* (Most Common Values, MCV). Давайте рассмотрим MCV и соответствующие дополнительные столбцы, которые пригодятся позже:

```
SELECT null_frac, n_distinct, most_common_vals, most_common_freqs FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

null_frac         | 0
n_distinct        | 676
most_common_vals  |
 {EJAAAA,BBAAAA,CRAAAA,FCAAAA,FEAAAA,GSAAAA,JOAAAA,MCAAAA,NAAAAA,WGAAAA}
most_common_freqs | {0.00333333,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003}
```

Так как значение `CRAAAA` оказалось в списке MCV, избирательность будет определяться просто соответствующим элементом в списке частот самых частых значений (Most Common Frequencies, MCF):

```
selectivity = mcf[3]
            = 0.003
```

Как и в предыдущем примере, оценка числа строк берётся как произведение мощности и избирательности `tenk1`:

```
rows = 10000 * 0.003
     = 30
```

Теперь рассмотрим тот же самый запрос, но с константой, которой нет в списке MCV:

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'xxx';

                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=15 width=244)
   Filter: (stringu1 = 'xxx'::name)
```

Это совершенно другая задача — как оценить избирательность значения, которого *нет* в списке MCV. При её решении используется факт отсутствия данного значения в списке в сочетании с частотой для каждого значения из списка MCV.

```
selectivity = (1 - sum(mvf))/(num_distinct - num_mcv)
            = (1 - (0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 +
                    0.003 + 0.003 + 0.003 + 0.003))/(676 - 10)
            = 0.0014559
```

Т. е. нужно сложить частоты значений из списка MCV, отнять полученное число от единицы, и полученное значение разделить на количество *остальных* уникальных значений. Эти вычисления основаны на предположении, что значения, которые не входят в список MCV, имеют равномерное распределение. Заметьте, что в данном примере нет неопределённых значений, поэтому о них беспокоиться не нужно (иначе их долю также пришлось бы вычитать из числителя). Оценка числа строк затем производится как обычно:

```
rows = 10000 * 0.0014559
     = 15  (округлённо)
```

Предыдущий пример с `unique1 < 1000` был большим упрощением того, что в действительности делает `scalarltsel`. Но после того, как мы увидели пример использования списка MCV, мы можем внести некоторые дополнения. Что касается самого примера, в нём все было правильно, поскольку `unique1` — это уникальный столбец, у него нет значений в списке MCV (очевидно, в данном случае нет значения, которое встречается чаще, чем какое-либо другое). Для неуникального столбца обычно создаётся как гистограмма, так и список MCV, при этом *гистограмма не включает значения, представленные в списке MCV*. Данный способ позволяет выполнить более точный подсчёт. В этой ситуации `scalarltsel` напрямую применяет условие «< 1000» к каждому значению списка MCV и суммирует частоты значений MCV, для которых условие является верным. Это даёт точную оценку избирательности для той части таблицы, которая содержит значения из списка MCV. Подобным же образом используется гистограмма для оценки избирательности для той части таблицы, которая не содержит значения из списка MCV, а затем эти две цифры складываются для оценки общей избирательности. Например, рассмотрим

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 < 'IAAAAA';

                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=3077 width=244)
   Filter: (stringu1 < 'IAAAAA'::name)
```

Мы уже видели данные списка MCV для `stringu1`, а это его гистограмма:

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

                                histogram_bounds
-------------------------------------------------------------------------------
 {AAAAAA,CQAAAA,FRAAAA,IBAAAA,KRAAAA,NFAAAA,PSAAAA,SGAAAA,VAAAAA,XLAAAA,ZZAAAA}
```

Проверяя список MCV, находим, что условие `stringu1 < 'IAAAAA'` соответствует первым шести записям, но не соответствует последним четырём, поэтому избирательность для значений, соответствующих значениям в списке MCV, такова:

```
selectivity = sum(relevant mvfs)
            = 0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003
            = 0.01833333
```

Сумма всех частот из списка MCF также сообщает нам, что общая часть представленной списком MCV совокупности записей равняется 0.03033333, и поэтому представленная гистограммой часть равняется 0.96966667 (в этом случае тоже нет неопределённых значений, иначе их пришлось бы также исключить). Видно, что значение `IAAAAA` попадает почти в конец третьего столбца гистограммы. Основываясь на простых предположениях относительно частоты различных символов, планировщик получает число 0.298387 для части значений, представленных в гистограмме, которые меньше чем `IAAAAA`. Затем объединяем оценки части значений из списка MCV и значений, не содержащихся в нём:

```
selectivity = mcv_selectivity + histogram_selectivity * histogram_fraction
            = 0.01833333 + 0.298387 * 0.96966667
            = 0.307669

rows        = 10000 * 0.307669
            = 3077  (округлённо)
```

В этом конкретном примере, корректировка со стороны списка MCV достаточно мала, потому что распределение значений столбца довольно плоское (статистика, показывающая конкретные значения как более распространённые, чаще всего получается вследствие статистической погрешности). В более типичном случае, когда некоторые значения являются значительно более распространёнными по сравнению с другими, этот более сложный метод повышает точность вследствие точного определения избирательности наиболее распространённых значений.

Теперь давайте рассмотрим случай с более чем одним условием в предложении `WHERE`:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000 AND stringu1 = 'xxx';

                                     QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=23.80..396.91 rows=1 width=244)
   Recheck Cond: (unique1 < 1000)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

Планировщик исходит из того, что два условия независимы, таким образом, отдельные значения избирательности можно перемножить:

```
selectivity = selectivity(unique1 < 1000) * selectivity(stringu1 = 'xxx')
            = 0.100697 * 0.0014559
            = 0.0001466

rows        = 10000 * 0.0001466
            = 1  (округлённо)
```

Заметьте, что число строк, которые предполагается вернуть через сканирование битового индекса, соответствует условию, используемому при работе индекса; это важно, так как влияет на оценку стоимости для последующих выборок из таблицы.

В заключение исследуем запрос, выполняющий соединение:

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
--------------------------------------------------------------------------------
 Nested Loop  (cost=4.64..456.23 rows=50 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.64..142.17 rows=50 width=244)
         Recheck Cond: (unique1 < 50)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.63 rows=50 width=0)
               Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..6.27 rows=1 width=244)
```

```
        Index Cond: (unique2 = t1.unique2)
```

Ограничение, накладываемое на `tenk1`, `unique1 < 50`, производится до соединения вложенным циклом. Это обрабатывается аналогично предыдущему примеру с диапазонным условием. На этот раз значение 50 попадает в первый столбец гистограммы `unique1`:

```
selectivity = (0 + (50 - bucket[1].min)/(bucket[1].max - bucket[1].min))/num_buckets
            = (0 + (50 - 0)/(993 - 0))/10
            = 0.005035

rows        = 10000 * 0.005035
            = 50  (округлённо)
```

Ограничение для соединения следующее `t2.unique2 = t1.unique2`. Здесь используется уже известный нам оператор `=`, однако функцию избирательности получаем из столбца `oprjoin` представления `pg_operator`, и эта функция — `eqjoinsel`. Функция `eqjoinsel` находит статистические данные как для `tenk2`, так и для `tenk1`:

```
SELECT tablename, null_frac,n_distinct, most_common_vals FROM pg_stats
WHERE tablename IN ('tenk1', 'tenk2') AND attname='unique2';

tablename  | null_frac | n_distinct | most_common_vals
-----------+-----------+------------+------------------
 tenk1     |         0 |         -1 |
 tenk2     |         0 |         -1 |
```

В этом случае нет данных MCV для `unique2`, потому что все значения будут уникальными. Таким образом, используется алгоритм, зависящий только от числа различающихся значений для обеих таблиц и от данных с неопределёнными значениями:

```
selectivity = (1 - null_frac1) * (1 - null_frac2) * min(1/num_distinct1, 1/
num_distinct2)
            = (1 - 0) * (1 - 0) / max(10000, 10000)
            = 0.0001
```

Т. е., вычитаем долю неопределённых значений из единицы для каждой таблицы и делим на максимальное из чисел различающихся значений. Количество строк, которое соединение, вероятно, сгенерирует, вычисляется как мощность декартова произведения двух входных значений, умноженная на избирательность:

```
rows = (outer_cardinality * inner_cardinality) * selectivity
     = (50 * 10000) * 0.0001
     = 50
```

Если бы имелись списки MCV для двух столбцов, функцией `eqjoinsel` использовалось бы прямое сравнение со списками MCV для определения общей избирательности той части данных, которая содержит значения списка MCV. Оценка остальной части данных при этом выполнялась бы представленным выше способом.

Заметьте, что здесь выводится для `inner_cardinality` значение 10000, то есть исходный размер `tenk2`. Если изучить вывод `EXPLAIN`, может показаться, что оценка количества строк вычисляется как 50 * 1, то есть число внешних строк умножается на ориентировочное число строк, получаемых при каждом внутреннем сканировании индекса в `tenk2`. Но это не так, ведь размер результата соединения оценивается до того, как выбирается конкретный план соединения. Если всё работает корректно, оба варианта вычисления этого размера должны давать один и тот же ответ, но из-за ошибок округления и других факторов иногда они значительно различаются.

Для интересующихся более подробной информацией: оценка размера таблицы (до выполнения условий в предложении `WHERE`) реализована в файле `src/backend/optimizer/util/plancat.c`. Основная логика для вычисления избирательности предложений находится в `src/backend/optimizer/path/clausesel.c`. Специфичные для отдельных операторов функции избирательности, в основном, расположены в `src/backend/utils/adt/selfuncs.c`.

65.2. Примеры многовариантной статистики

65.2.1. Функциональные зависимости

Многовариантную корреляцию можно продемонстрировать на очень простом наборе данных — таблице с двумя столбцами, содержащими одинаковые значения:

```
CREATE TABLE t (a INT, b INT);
INSERT INTO t SELECT i % 100, i % 100 FROM generate_series(1, 10000) s(i);
ANALYZE t;
```

Как рассказывается в Разделе 14.2, планировщик может определить мощность `t`, исходя из числа страниц и строк, полученного из `pg_class`:

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 't';

 relpages | reltuples
----------+-----------
       45 |     10000
```

Распределение данных очень простое: в каждом столбце содержится всего 100 различных значений, равномерно распределённых.

Следующий пример показывает результат оценивания условия `WHERE` по столбцу `a`:

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..170.00 rows=100 width=8) (actual rows=100 loops=1)
   Filter: (a = 1)
   Rows Removed by Filter: 9900
```

Планировщик рассматривает условие и определяет, что его избирательность равна 1%. Сравнивая эту оценку и фактическое число строк, мы видим, что оценка очень точна (на самом деле абсолютна точна, так как таблица очень маленькая). Если изменить условие `WHERE`, чтобы использовался столбец `b`, будет получен такой же план. Но посмотрите, что получится, если мы применим одинаковое условие к двум столбцам, объединив их оператором `AND`:

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b = 1;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=1 width=8) (actual rows=100 loops=1)
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
```

Планировщик оценивает избирательность каждого условия индивидуально, и получает ту же оценку в 1%, что и выше. Затем он предполагает, что условия независимы, так что он перемножает избирательности и выдаёт окончательную оценку избирательности, равную всего 0.01%. Это значительная недооценка, так как фактическое число строк, соответствующих условию, (100) на два порядка больше.

Эту проблему можно решить, создав объект статистики, который укажет команде `ANALYZE` вычислить многовариантную статистику функциональной зависимости по двум столбцам:

```
CREATE STATISTICS stts (dependencies) ON a, b FROM t;
ANALYZE t;
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b = 1;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=100 width=8) (actual rows=100 loops=1)
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
```

65.2.2. Многовариантное число различных значений

Подобная проблема возникает с оценкой мощности наборов с несколькими столбцами, например, с оценкой числа групп, которые могут быть выданы предложением `GROUP BY`. Когда в `GROUP BY` указан один столбец, оценка числа различных значений (которую можно увидеть как ожидаемое число строк, выдаваемое узлом HashAggregate) очень точная:

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a;
                                     QUERY PLAN

--------------------------------------------------------------------------------------
 HashAggregate  (cost=195.00..196.00 rows=100 width=12) (actual rows=100 loops=1)
   Group Key: a
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=4) (actual rows=10000
 loops=1)
```

Но оценка числа групп в запросе с двумя столбцами в `GROUP BY` без многовариантной статистики, как и в предыдущем примере, отличается от правильной на порядок:

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a, b;
                                     QUERY PLAN

-----------------------------------------------------------------------------------------------
 HashAggregate  (cost=220.00..230.00 rows=1000 width=16) (actual rows=100 loops=1)
   Group Key: a, b
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=8) (actual rows=10000
 loops=1)
```

Если переопределить объект статистики, чтобы он включал подсчёт числа различных значений для двух столбцов, оценка станет гораздо лучше:

```
DROP STATISTICS stts;
CREATE STATISTICS stts (dependencies, ndistinct) ON a, b FROM t;
ANALYZE t;
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a, b;
                                     QUERY PLAN

-----------------------------------------------------------------------------------------------
 HashAggregate  (cost=220.00..221.00 rows=100 width=16) (actual rows=100 loops=1)
   Group Key: a, b
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=8) (actual rows=10000
 loops=1)
```

65.3. Статистика планировщика и безопасность

Доступ к таблице `pg_statistic` разрешён только суперпользователям, так что обычные пользователи не могут получить из неё сведения о содержимом таблиц других пользователей. Но некоторые функции оценки избирательности будут использовать пользовательский оператор (оператор, фигурирующий в запросе, или связанный) для анализа сохранённой статистики. Например, чтобы определить применимость сохранённого самого частого значения, функция оценки избирательности должна задействовать соответствующий оператор `=` для сравнения константы в запросе с этим сохранённым значением. Таким образом, данные `pg_statistic` в принципе могут передаваться пользовательским операторам. А особым образом сконструированный оператор может выводить наружу передаваемые ему операнды преднамеренно (например, записывая их в журнал или помещая в другую таблицу) либо непреднамеренно (показывая их значения в сообщениях об ошибках). В любом случае это даёт возможность пользователю, не имеющему доступа к таблице `pg_statistic`, увидеть содержащиеся в ней данные.

Для предотвращения этого все встроенные функции оценки избирательности действуют по следующим правилам. Чтобы сохранённая статистика могла использоваться при планировании

запроса, текущий пользователь должен иметь либо право SELECT для таблицы или задействованных столбцов, либо у оператора должна быть характеристика LEAKPROOF (точнее, она должна быть у функции, реализующей этот оператор). В противном случае оценка избирательности будет осуществляться так, как если бы статистики не было вовсе, и планировщик продолжит работу с общими или вторичными предположениями.

Если пользователь не имеет требуемого права доступа к таблице или столбцам, то во многих случаях при выполнении запроса в конце концов возникнет ошибка «нет доступа», так что этот механизм будет незаметен на практике. Но если пользователь читает данные из представления с барьером безопасности, планировщик может захотеть проверить статистику нижележащей таблицы, которая недоступна пользователю непосредственно. В этом случае оператор должен быть герметичным; иначе статистика не будет использоваться. Это не будет иметь внешних проявлений кроме того, что план запроса может быть неоптимальным. В случае подозрений, что вы столкнулись с этим, попробуйте запустить запрос от имени пользователя с расширенными правами и проверьте, не выбирается ли другой план запроса.

Это ограничение применяется только тогда, когда планировщику может потребоваться выполнить пользовательский оператор с одним или несколькими значениями из pg_statistic. При этом планировщику разрешено использовать общую статистическую информацию, например, процент значений NULL или количество различных значений в столбце, вне зависимости от прав доступа.

Реализуемые в дополнительных расширениях функции оценки избирательности, которые могут обращаться к статистике, вызывая пользовательские операторы, должны следовать тем же правилам безопасности.

Часть VIII. Приложения

Приложение A. Коды ошибок Postgres Pro

Всем сообщениям, которые выдаёт сервер Postgres Pro, назначены пятисимвольные коды ошибок, соответствующие кодам «SQLSTATE», описанным в стандарте SQL. Приложения, которые должны знать, какое условие ошибки имело место, обычно проверяют код ошибки и только потом обращаются к текстовому сообщению об ошибке. Коды ошибок, скорее всего, не изменятся от выпуска к выпуску Postgres Pro, и они не меняются при локализации как сообщения об ошибках. Заметьте, что отдельные, но не все коды ошибок, которые выдаёт Postgres Pro, определены стандартом SQL; некоторые дополнительные коды ошибок для условий, не описанных стандартом, были добавлены независимо или позаимствованы из других баз данных.

Согласно стандарту, первые два символа кода ошибки обозначают класс ошибок, а последние три символа обозначают определённое условие в этом классе. Таким образом, приложение, не знающее значение определённого кода ошибки, всё же может понять, что делать, по классу ошибки.

В Таблице A.1 перечислены все коды ошибок, определённые в Postgres Pro Standard 10.23.1. (Некоторые коды в настоящее время не используются, хотя они определены в стандарте SQL.) Также показаны классы ошибок. Для каждого класса ошибок имеется «стандартный» код ошибки с последними тремя символами `000`. Этот код выдаётся только для таких условий ошибок, которые относятся к некоторому классу, но не имеют более определённого кода.

Символ, указанный в столбце «Имя условия», определяет условие в PL/pgSQL. Имена условий могут записываться в верхнем или нижнем регистре. (Заметьте, что PL/pgSQL, в отличие от ошибок, не распознаёт предупреждения; то есть классы 00, 01 и 02.)

Для некоторых типов ошибок сервер сообщает имя объекта базы данных (таблица, столбец таблицы, тип данных или ограничение), связанного с ошибкой; например, имя уникального ограничения, вызвавшего ошибку `unique_violation`. Такие имена передаются в отдельных полях сообщения об ошибке, чтобы приложениям не пришлось извлекать его из возможно локализованного текста ошибки для человека. На момент выхода PostgreSQL 9.3 полностью охватывались только ошибки класса SQLSTATE 23 (нарушения ограничений целостности), но в будущем должны быть охвачены и другие классы.

Таблица A.1. Коды ошибок Postgres Pro

Код ошибки	Имя условия
Класс 00 — Успешное завершение	
00000	successful_completion
Класс 01 — Предупреждение	
01000	warning
0100C	dynamic_result_sets_returned
01008	implicit_zero_bit_padding
01003	null_value_eliminated_in_set_function
01007	privilege_not_granted
01006	privilege_not_revoked
01004	string_data_right_truncation
01P01	deprecated_feature
Класс 02 — Нет данных (это также класс предупреждений согласно стандарту SQL)	
02000	no_data

Код ошибки	Имя условия
02001	no_additional_dynamic_result_sets_returned
Класс 03 — SQL-оператор ещё не завершён	
03000	sql_statement_not_yet_complete
Класс 08 — Исключение, связанное с подключением	
08000	connection_exception
08003	connection_does_not_exist
08006	connection_failure
08001	sqlclient_unable_to_establish_sqlconnection
08004	sqlserver_rejected_establishment_of_sqlconnection
08007	transaction_resolution_unknown
08P01	protocol_violation
Класс 09 — Исключение с действием триггера	
09000	triggered_action_exception
Класс 0A — Неподдерживаемая функциональность	
0A000	feature_not_supported
Класс 0B — Неверное начало транзакции	
0B000	invalid_transaction_initiation
Класс 0F — Исключение с указателем на данные	
0F000	locator_exception
0F001	invalid_locator_specification
Класс 0L — Неверный праводатель	
0L000	invalid_grantor
0LP01	invalid_grant_operation
Класс 0P — Неверное указание роли	
0P000	invalid_role_specification
Класс 0Z — Исключение диагностики	
0Z000	diagnostics_exception
0Z002	stacked_diagnostics_accessed_without_active_handler
Класс 20 — Case не найден	
20000	case_not_found
Класс 21 — Нарушение количества	
21000	cardinality_violation
Класс 22 — Исключение в данных	
22000	data_exception
2202E	array_subscript_error
22021	character_not_in_repertoire
22008	datetime_field_overflow

Код ошибки	Имя условия
22012	division_by_zero
22005	error_in_assignment
2200B	escape_character_conflict
22022	indicator_overflow
22015	interval_field_overflow
2201E	invalid_argument_for_logarithm
22014	invalid_argument_for_ntile_function
22016	invalid_argument_for_nth_value_function
2201F	invalid_argument_for_power_function
2201G	invalid_argument_for_width_bucket_function
22018	invalid_character_value_for_cast
22007	invalid_datetime_format
22019	invalid_escape_character
2200D	invalid_escape_octet
22025	invalid_escape_sequence
22P06	nonstandard_use_of_escape_character
22010	invalid_indicator_parameter_value
22023	invalid_parameter_value
2201B	invalid_regular_expression
2201W	invalid_row_count_in_limit_clause
2201X	invalid_row_count_in_result_offset_clause
2202H	invalid_tablesample_argument
2202G	invalid_tablesample_repeat
22009	invalid_time_zone_displacement_value
2200C	invalid_use_of_escape_character
2200G	most_specific_type_mismatch
22004	null_value_not_allowed
22002	null_value_no_indicator_parameter
22003	numeric_value_out_of_range
2200H	sequence_generator_limit_exceeded
22026	string_data_length_mismatch
22001	string_data_right_truncation
22011	substring_error
22027	trim_error
22024	unterminated_c_string
2200F	zero_length_character_string
22P01	floating_point_exception
22P02	invalid_text_representation

Код ошибки	Имя условия
22P03	invalid_binary_representation
22P04	bad_copy_file_format
22P05	untranslatable_character
2200L	not_an_xml_document
2200M	invalid_xml_document
2200N	invalid_xml_content
2200S	invalid_xml_comment
2200T	invalid_xml_processing_instruction
Класс 23 — Нарушение ограничения целостности	
23000	integrity_constraint_violation
23001	restrict_violation
23502	not_null_violation
23503	foreign_key_violation
23505	unique_violation
23514	check_violation
23P01	exclusion_violation
Класс 24 — Неверное состояние курсора	
24000	invalid_cursor_state
Класс 25 — Неверное состояние транзакции	
25000	invalid_transaction_state
25001	active_sql_transaction
25002	branch_transaction_already_active
25008	held_cursor_requires_same_isolation_level
25003	inappropriate_access_mode_for_branch_transaction
25004	inappropriate_isolation_level_for_branch_transaction
25005	no_active_sql_transaction_for_branch_transaction
25006	read_only_sql_transaction
25007	schema_and_data_statement_mixing_not_supported
25P01	no_active_sql_transaction
25P02	in_failed_sql_transaction
25P03	idle_in_transaction_session_timeout
Класс 26 — Неверное имя SQL-оператора	
26000	invalid_sql_statement_name
Класс 27 — Нарушение при изменении данных в триггере	
27000	triggered_data_change_violation
Класс 28 — Неверное указание авторизации	

Код ошибки	Имя условия
28000	invalid_authorization_specification
28P01	invalid_password
Класс 2B — Зависимые описания привилегий всё ещё существуют	
2B000	dependent_privilege_descriptors_still_exist
2BP01	dependent_objects_still_exist
Класс 2D — Неверное завершение транзакции	
2D000	invalid_transaction_termination
Класс 2F — Исключение в подпрограмме SQL	
2F000	sql_routine_exception
2F005	function_executed_no_return_statement
2F002	modifying_sql_data_not_permitted
2F003	prohibited_sql_statement_attempted
2F004	reading_sql_data_not_permitted
Класс 34 — Неверное имя курсора	
34000	invalid_cursor_name
Класс 38 — Исключение во внешней подпрограмме	
38000	external_routine_exception
38001	containing_sql_not_permitted
38002	modifying_sql_data_not_permitted
38003	prohibited_sql_statement_attempted
38004	reading_sql_data_not_permitted
Класс 39 — Исключение при вызове внешней подпрограммы	
39000	external_routine_invocation_exception
39001	invalid_sqlstate_returned
39004	null_value_not_allowed
39P01	trigger_protocol_violated
39P02	srf_protocol_violated
39P03	event_trigger_protocol_violated
Класс 3B — Исключение точки сохранения	
3B000	savepoint_exception
3B001	invalid_savepoint_specification
Класс 3D — Неверное имя каталога	
3D000	invalid_catalog_name
Класс 3F — Неверное имя схемы	
3F000	invalid_schema_name
Класс 40 — Откат транзакции	
40000	transaction_rollback
40002	transaction_integrity_constraint_violation
40001	serialization_failure

Код ошибки	Имя условия
40003	statement_completion_unknown
40P01	deadlock_detected
Класс 42 — Ошибка синтаксиса или нарушение правила доступа	
42000	syntax_error_or_access_rule_violation
42601	syntax_error
42501	insufficient_privilege
42846	cannot_coerce
42803	grouping_error
42P20	windowing_error
42P19	invalid_recursion
42830	invalid_foreign_key
42602	invalid_name
42622	name_too_long
42939	reserved_name
42804	datatype_mismatch
42P18	indeterminate_datatype
42P21	collation_mismatch
42P22	indeterminate_collation
42809	wrong_object_type
428C9	generated_always
42703	undefined_column
42883	undefined_function
42P01	undefined_table
42P02	undefined_parameter
42704	undefined_object
42701	duplicate_column
42P03	duplicate_cursor
42P04	duplicate_database
42723	duplicate_function
42P05	duplicate_prepared_statement
42P06	duplicate_schema
42P07	duplicate_table
42712	duplicate_alias
42710	duplicate_object
42702	ambiguous_column
42725	ambiguous_function
42P08	ambiguous_parameter
42P09	ambiguous_alias
42P10	invalid_column_reference
42611	invalid_column_definition

Код ошибки	Имя условия
42P11	invalid_cursor_definition
42P12	invalid_database_definition
42P13	invalid_function_definition
42P14	invalid_prepared_statement_definition
42P15	invalid_schema_definition
42P16	invalid_table_definition
42P17	invalid_object_definition
Класс 44 — Нарушение WITH CHECK OPTION	
44000	with_check_option_violation
Класс 53 — Нехватка ресурсов	
53000	insufficient_resources
53100	disk_full
53200	out_of_memory
53300	too_many_connections
53400	configuration_limit_exceeded
Класс 54 — Превышение ограничения программы	
54000	program_limit_exceeded
54001	statement_too_complex
54011	too_many_columns
54023	too_many_arguments
Класс 55 — Объект не в требуемом состоянии	
55000	object_not_in_prerequisite_state
55006	object_in_use
55P02	cant_change_runtime_param
55P03	lock_not_available
Класс 57 — Вмешательство оператора	
57000	operator_intervention
57014	query_canceled
57P01	admin_shutdown
57P02	crash_shutdown
57P03	cannot_connect_now
57P04	database_dropped
Класс 58 — Ошибка системы (ошибка, внешняя по отношению к PostgreSQL)	
58000	system_error
58030	io_error
58P01	undefined_file
58P02	duplicate_file
Класс 72 — Ошибка снимка	
72000	snapshot_too_old
Класс F0 — Ошибка файла конфигурации	

Код ошибки	Имя условия
F0000	config_file_error
F0001	lock_file_exists
Класс HV — Ошибка обёртки сторонних данных (SQL/MED)	
HV000	fdw_error
HV005	fdw_column_name_not_found
HV002	fdw_dynamic_parameter_value_needed
HV010	fdw_function_sequence_error
HV021	fdw_inconsistent_descriptor_information
HV024	fdw_invalid_attribute_value
HV007	fdw_invalid_column_name
HV008	fdw_invalid_column_number
HV004	fdw_invalid_data_type
HV006	fdw_invalid_data_type_descriptors
HV091	fdw_invalid_descriptor_field_identifier
HV00B	fdw_invalid_handle
HV00C	fdw_invalid_option_index
HV00D	fdw_invalid_option_name
HV090	fdw_invalid_string_length_or_buffer_length
HV00A	fdw_invalid_string_format
HV009	fdw_invalid_use_of_null_pointer
HV014	fdw_too_many_handles
HV001	fdw_out_of_memory
HV00P	fdw_no_schemas
HV00J	fdw_option_name_not_found
HV00K	fdw_reply_handle
HV00Q	fdw_schema_not_found
HV00R	fdw_table_not_found
HV00L	fdw_unable_to_create_execution
HV00M	fdw_unable_to_create_reply
HV00N	fdw_unable_to_establish_connection
Класс P0 — Ошибка PL/pgSQL	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows
P0004	assert_failure
Класс XX — Внутренняя ошибка	
XX000	internal_error

Код ошибки	**Имя условия**
XX001	data_corrupted
XX002	index_corrupted

Приложение B. Поддержка даты и времени

Postgres Pro использует внутренний эвристический анализатор для поддержки всех значений даты и времени. Дата и время вводятся как строки и разделяются на различные поля, при этом предварительно определяется, какого рода информация содержится в конкретном поле. Каждое поле интерпретируется и получает числовое значение, игнорируется или отклоняется. Анализатор содержит внутренние справочные таблицы для текстовых полей, включая месяцы, дни недели и часовые пояса.

Данное приложение включает информацию о содержании справочных таблиц и описывает этапы, необходимые анализатору для распознавания даты и времени.

B.1. Интерпретация данных даты и времени

Строки с датой/временем разбираются при вводе по следующему алгоритму.

1. Разделить входную строку на фрагменты и определить каждый фрагмент как строку, время, часовой пояс или цифру.

    a. Если числовой фрагмент содержит двоеточие (`:`), значит эта строка представляет время. Включаются все последующие цифры и двоеточия.

    b. Если числовой фрагмент содержит тире (`-`), косую черту (`/`) или две и более точек (`.`), то это строка даты, которая, возможно, включает название месяца. Если фрагмент даты уже встречался, он интерпретируется как название часового пояса (например, `America/New_York`).

    c. Если этот фрагмент является лишь числом, он представляет собой отдельное поле или составную дату ISO 8601 (например, `19990113` для 13 января 1999 года) или время (например, `141516` для 14:15:16).

    d. Если фрагмент начинается с плюса (`+`) или минуса (`-`), то это или числовой часовой пояс или специальное поле.

2. Если фрагмент содержит только буквы, сопоставить его с возможными строками:

    a. Проверить, не совпадает ли фрагмент с известной аббревиатурой часового пояса. Эти аббревиатуры считываются из файла конфигурации, описанного в Разделе B.4.

    b. Если фрагмент не найден, проверить во внутренней таблице, не совпадает ли он со специальной строкой (например, `today`), днём недели (например, `Thursday`), месяцем (например, `January`) или игнорируемым словом (например, `at`, `on`).

    c. Если фрагмент всё же не найден, выдать ошибку.

3. Когда фрагмент является числом или числовым полем:

    a. Если получено восемь или шесть цифр и никакое другое поле даты ранее не было прочитано, интерпретировать их как «составленную дату» (например, `19990118` или `990118`). Такая дата интерпретируется как `ГГГГММДД` или `ГГММДД`.

    b. Если фрагмент представляет собой трёхзначное число, и год уже был прочитан, интерпретировать как день года.

    c. Если это четыре или шесть цифр и год уже был прочитан, интерпретировать как время (`ЧЧММ` или `ЧЧММСС`).

    d. Если найдены три или более цифр, а поля даты ещё не были найдены, интерпретировать как год (это ведёт к установке порядка гг-мм-дд для оставшихся полей даты).

e. В противном случае подразумевается, что порядок сортировки полей даты определяется значением `DateStyle`: мм-дд-гг, дд-мм-гг или гг-мм-дд. Выдать ошибку, если оказалось, что поле месяца или дня вышло за пределы диапазона.

4. Если указан год до н. э., отнять год и добавить единицу для внутреннего хранения. (В григорианском календаре отсутствует нулевой год, поэтому 1 год до н. э. становится нулевым.)

5. Если год до н. э. не был указан, и если поле года имело два разряда, установить для записи года четыре разряда. Если поле меньше 70, добавить 2000, в противном случае добавить 1900.

> **Подсказка**
>
> Годы с 1 по 99 н. э. по григорианскому календарю могут вводится при помощи четырёхзначного числа с начальными нулями (например, `0099` это год 99 н. э.).

B.2. Обработка недопустимых или неоднозначных значений даты/времени

Обычно, если строка даты/времени синтаксически корректна, но содержит поля со значениями вне допустимого диапазона, выдаётся ошибка. Например, не будет принята строка с числом 31 февраля.

Однако с учётом перевода часов на летнее время визуально корректная строка с датой/временем может указывать не несуществующий или неоднозначный момент времени. Такие значения тем не менее принимаются; для разрешения неоднозначности учитывается смещение от UTC. Например, в предположении, что значение TimeZone — `America/New_York`, рассмотрите следующий пример:

```
=> SELECT '2018-03-11 02:30'::timestamptz;
      timestamptz
------------------------
 2018-03-11 03:30:00-04
(1 row)
```

Так как в этот день в данном часовом поясе стрелки часов переводились вперёд, по гражданским часам не существовал момент времени 2:30; они перескочили с 2:00 (EST) на 3:00 (EDT). Postgres Pro воспринимает заданное время, как если бы оно было стандартным временем (в часовом поясе UTC-5), в результате чего оно преобразуется в 3:30AM (в часовом поясе EDT или UTC-4).

И наоборот, рассмотрите поведение в случае перевода времени назад:

```
=> SELECT '2018-11-04 01:30'::timestamptz;
      timestamptz
------------------------
 2018-11-04 01:30:00-05
(1 row)
```

В этот день возможны две интерпретации времени 1:30; сначала было 1:30 в часовом поясе EDT, а час спустя, после перевода часов назад с 2:00 на 1:00 и смены часового пояса с EDT на EST, наступило 1:30 в часовом поясе EST. Так же и в этом случае Postgres Pro интерпретирует заданное время, как если бы оно было стандартным (UTC-5). Мы можем выбрать другой момент, указав часовой пояс летнего времени:

```
=> SELECT '2018-11-04 01:30 EDT'::timestamptz;
      timestamptz
------------------------
 2018-11-04 01:30:00-04
(1 row)
```

Точное правило, применяемое в таких случаях, звучит так: несуществующий момент времени, попадающий в интервал перевода часов вперёд, привязывается к смещению от UTC,

действовавшему в данном часовом поясе до перевода, а неоднозначный момент времени, попадающий в оба интервала при переводе часов назад, привязывается к смещению от UTC, действующему после перевода. Для большинства часовых поясов это равносильно правилу «в случае неясности предпочитать интерпретацию со стандартным временем».

В любом случае смещение от UTC, связанное с меткой времени, можно задать явно, добавив либо числовое смещение, либо аббревиатуру часового пояса, которой соответствует некоторое фиксированное смещение. Вышеприведённое правило применяется, только когда необходимо вычислить смещение от UTC для часового пояса, в котором оно меняется.

B.3. Ключевые слова для обозначения даты и времени

Таблица B.1 показывает фрагменты, которые распознаются как названия месяцев.

Таблица B.1. Названия месяцев

Месяц	Аббревиатуры
January	Jan
February	Feb
March	Mar
April	Apr
May	
June	Jun
July	Jul
August	Aug
September	Sep, Sept
October	Oct
November	Nov
December	Dec

Таблица B.2 показывает фрагменты, которые распознаются как названия дней недели.

Таблица B.2. Названия дней недели

День	Аббревиатуры
Sunday	Sun
Monday	Mon
Tuesday	Tue, Tues
Wednesday	Wed, Weds
Thursday	Thu, Thur, Thurs
Friday	Fri
Saturday	Sat

Таблица B.3 показывает фрагменты, которые выполняют различные функции модификаторов.

Таблица B.3. Модификаторы поля даты/времени

Идентификатор	Описание
`AM`	Время до 12:00
`AT`	Игнорируется
`JULIAN`, `JD`, `J`	Следующее поле является юлианской датой

Идентификатор	Описание
ON	Игнорируется
PM	Время 12:00 и более позднее
T	Следующее поле указывает на время

B.4. Файлы конфигурации даты/времени

Поскольку аббревиатуры часовых поясов недостаточно стандартизированы, Postgres Pro предлагает средства для определения набора аббревиатур, принимаемых сервером. Параметром выполнения timezone_abbreviations определяется активный набор аббревиатур. Хотя данный параметр может быть изменён любым пользователем базы данных, возможные значения для него контролируются администратором базы данных и являются именами конфигурационных файлов, хранящихся в .../share/timezonesets/ каталога установки. Добавляя или изменяя файлы в этом каталоге, администратор может определить местную специфику выбора аббревиатур часовых поясов.

Значение timezone_abbreviations может быть установлено в любое имя файла, находящегося в .../share/timezonesets/, если имя файла состоит только из букв. (Запрет на использование небуквенных символов в timezone_abbreviations делает невозможным чтение файлов, находящихся вне заданного каталога, а также резервных файлов редактора и прочих внешних файлов.)

Файл аббревиатур часовых поясов может содержать пустые строки и комментарии, начинающиеся с #. Строки, не имеющие комментариев, должны иметь один из следующих форматов:

```
аббревиатура_пояса смещение
аббревиатура_пояса смещение D
аббревиатура_пояса имя_часового_пояса
@INCLUDE имя_файла
@OVERRIDE
```

Поле *аббревиатура_пояса* лишь задаёт определяемую аббревиатуру. *Смещение* — это целое число, задающее эквивалентное смещение от UTC в секундах, положительное — к востоку от Гринвичского меридиана, а отрицательное — к западу. Например, -18000 означало бы пять часов к западу от Гринвича, или Североамериканское восточное время. D указывает, что название пояса представляет местное летнее, а не поясное время.

В качестве альтернативы может быть задано *имя_часового_пояса*, определённое в базе данных часовых поясов IANA. В этом случае система обращается к определению пояса, чтобы выяснить, используется или использовалась ли аббревиатура для этого пояса, и если да, действовать будет соответствующее значение — то есть, значение, действовавшее в указанный момент времени, или действовавшее непосредственно перед ним, если текущее на тот момент неизвестно, либо самое старое значение, если первое определение появилось после этого момента. Это поведение важно для тех аббревиатур, значение которых менялось в ходе истории. Также допускается определение аббревиатуры через имя часового пояса, с которым эта аббревиатура не связана; в таком случае использование аббревиатуры равнозначно написанию просто имени пояса.

> **Подсказка**
>
> Простое целочисленное *смещение* предпочтительнее использовать, когда определяется аббревиатура, смещение которой от UTC никогда не менялось, так как обрабатывать подобные аббревиатуры гораздо легче, чем те, что требуют обращения к определению часового пояса.

Использование @INCLUDE позволяет включить другой файл в каталоге .../share/timezonesets/. Включение может быть вложенным до ограниченной глубины.

Использование `@OVERRIDE` указывает, что последующие записи в файле могут переопределять предыдущие (как правило, это записи, полученные из включённых файлов). Без этого указания конфликтующие определения аббревиатуры одного и того же часового пояса считаются ошибкой.

При установке без внесения изменений, файл `Default` содержит все неконфликтующие аббревиатуры часовых поясов для большей части земного шара. Дополнительные файлы `Australia` и `India` предоставляются для данных регионов. Эти файлы сначала включают файл `Default`, а затем добавляют и изменяют аббревиатуры по мере необходимости.

В качестве справочной информации стандартная установка также содержит файлы `Africa.txt`, `America.txt` и т. д., содержащие информацию о каждой используемой аббревиатуре часового пояса, включённой в базу данных часовых поясов IANA. Определения названий часовых поясов, находящиеся в этих файлах, можно копировать и помещать в файл с нестандартной конфигурацией по мере необходимости. Заметьте, что данные файлы нельзя указывать непосредственно в значении `timezone_abbreviations`, так как их имена включают точку.

Примечание

Если при чтении набора аббревиатур часовых поясов возникает ошибка, новое значение не применяется и сохраняется старый набор. Если ошибка возникает при запуске базы данных, происходит сбой.

Внимание

Аббревиатуры часового пояса, определённые в файле конфигурации, переопределяют не относящиеся к часовому поясу значения, встроенные в Postgres Pro. Например, файл конфигурации `Australia` определяет `SAT` (для Южноавстралийского стандартного времени, South Australian Standard Time). Когда этот файл активен, `SAT` не будет распознаваться как сокращение слова «суббота».

Внимание

Если вы модифицируете файлы в `.../share/timezonesets/`, вы можете сами выполнить резервное копирование, так как обычный дамп базы данных не содержит этот каталог.

B.5. Указание часовых поясов в стиле POSIX

Postgres Pro может воспринимать указание часового пояса, записанное по правилам стандарта POSIX, в соответствии с которыми, в частности, задаётся переменная окружения `TZ`. Указания часовых поясов в стиле POSIX не удовлетворяют всем условиям, продиктованным сложной историей часовых поясов в реальном мире, но иногда использование этих указаний бывает оправданным.

Указание часового пояса в стиле POSIX выглядит так:

STD смещение [*DST* [*смещение_летнего_времени*] [, *правило*]]

(Для наглядности между полями добавлены пробелы, но в реальном указании их не должно быть.) В нём содержатся следующие поля:

- *STD* — аббревиатура, используемая для стандартного часового пояса.
- *смещение* — стандартное смещение пояса от UTC.
- *DST* — аббревиатура часового пояса при переходе на летнее время. Если это поле и следующие за ним опущены, для часового пояса будет использоваться фиксированное смещение от UTC без учёта перехода на летнее время.

- *смещение_летнего_времени* — смещение часового пояса от UTC при переходе на летнее время. Это поле обычно опускается, так как по умолчанию это смещение равно стандартному *смещению* минус один час, что практически всегда имеет место на практике.
- *правило* — определяет, как должен учитываться переход на летнее время, в соответствии с представленным ниже описанием.

В этой записи аббревиатура часового пояса может задаваться набором букв, например `EST`, или произвольной строкой, заключённой в угловые скобки, например `<UTC-05>`. Заметьте, что показанные здесь аббревиатуры используются только при выводе и только в некоторых выходных форматах. Распознаваемые во входных значениях даты/времени аббревиатуры часовых поясов описаны в Разделе B.4.

В полях смещений задаётся сдвиг от UTC в часах и, возможно, минутах и секундах. Они имеют формат *чч*[:*мм*[:*сс*]] и могут содержать спереди знак (+ или −). Положительный знак имеют часовые пояса *к западу* от Гринвича. (Заметьте, что это противоположно соглашению ISO-8601, используемому в других областях Postgres Pro.) Компонент *чч* может содержать одну или две цифры, а *мм* и *сс* (если они задаются) должны состоять ровно из двух цифр.

Поле, задающее *правило* перехода на летнее время, имеет следующий формат:

дата_перехода [/ *время_перехода*] , *дата_возврата* [/ *время_возврата*]

(Как и ранее, реальное указание не должно содержать пробелы.) Поля *дата_перехода* и *время_перехода* определяют момент перехода на летнее время, а поля *дата_возврата* и *время_возврата* определяют дату возврата к поясному времени. (В некоторых регионах, в частности, в Южном полушарии, вторая дата в календаре может предшествовать первой.) Даты могут задаваться в одном из следующих форматов:

n

Простое целое, обозначающее день года, с нумерацией от нуля до 364, или 365 в високосном году.

Jn

В этой записи *n* обозначает номер дня от 1 до 365, а 29 февраля пропускается, даже когда фактически присутствует. (Таким образом, смены часового пояса, происходящие 29 февраля, в этой записи выразить нельзя. Однако после февраля дни имеют одинаковые номера и в обычные, и в високосные годы, так что эта запись обычно полезнее тем, что позволяет выразить одним числом фиксированную дату перевода часов.)

M*m*.*n*.*d*

Эта форма задаёт дату перехода, которая всегда назначается в определённый месяц и день недели. Здесь *m* обозначает номер месяца от 1 до 12, а *n* — номер повторения в этом месяце дня недели, заданного полем *d*. В качестве *n* может задаваться число от 1 до 4 либо число 5, выбирающее последнюю дату с заданным днём недели в этом месяце (она может быть фактически четвёртой или пятой). В качестве *d* задаётся число от 0 до 6, обозначающее день недели (0 — воскресенье). Например, запись `M3.2.0` расшифровывается как «второе воскресенье марта».

> **Примечание**
>
> Для описания многих существующих правил перехода на летнее время обычно достаточно формата `M`. Но заметьте, что ни одна из этих вариаций не позволяет описать изменения правил перевода часов. Поэтому на практике для правильной интерпретации значений времени, относящихся к прошлому, необходимы исторические данные, хранящиеся в привязке к именованным часовым поясам (в базе данных IANA с информацией о часовых поясах).

Поля времени в описании правила перехода имеют тот же формат, что и поля смещений, описанные ранее, за исключением того, что в них не может быть знака. Они определяют текущее местное время, в которое производится перевод часов. По умолчанию временем перевода часов считается `02:00:00`.

Если задаётся аббревиатура для летнего времени, но *правило* перехода не задано, Postgres Pro пытается определить время перехода, обратившись к файлу `posixrules`, относящемуся к базе данных часовых поясов IANA. Этот файл позволяет определить полное указание часового пояса, но из этого указания берутся только правила перевода часов, но не смещения от UTC. Как правило, содержимое этого файла совпадает с содержимым файла `US/Eastern`, поэтому указание часового пояса в стиле POSIX подразумевает использование американских правил перехода на летнее время. При необходимости это можно поменять, заменив файл `posixrules`.

> **Примечание**
>
> Механизм использования файла `posixrules` переведён в IANA в разряд устаревших, так что в будущем он может быть удалён. Но в нём есть один дефект, который вряд ли будет устранён до момента его удаления. Вследствие этого дефекта к датам после 2038 г. невозможно применить правила перехода на летнее время.

В случае отсутствия файла `posixrules` выбирается правило `M3.2.0,M11.1.0`, соответствующее правилу перевода часов, действующему в США в 2020 г. (то есть переход на летнее время производится во второе воскресенье марта, а назад — в первое воскресенье ноября; часы переводятся в 2:00).

Например, запись `CET-1CEST,M3.5.0,M10.5.0/3` отражает действующую в 2020 г. практику перевода часов в Париже. Это указание говорит, что стандартное поясное время имеет аббревиатуру `CET` и оно смещено на час вперёд (к востоку) от UTC; летнее время имеет аббревиатуру `CEST` и смещено вперед от UTC на два часа (это определяется неявно). Часы переводятся на летнее время в последнее воскресенье марта в 2:00 CET, а на зимнее — в последнее воскресенье октября в 3:00 CEST.

Названия четырёх часовых поясов `EST5EDT`, `CST6CDT`, `MST7MDT` и `PST8PDT` выглядят как определяющие часовые пояса в стиле POSIX, но в реальности по историческим причинам они воспринимаются как имена часовых поясов, наравне с другими, так как в базе данных IANA есть файлы с такими именами. Практическое следствие этого состоит в том, что для данных имён может быть получена историческая информация о действовавших правилах перехода на летнее время в США, которую не может дать обычное указание в стиле POSIX ввиду отсутствия подходящего файла `posixrules`.

Имейте в виду, что ошибиться в указании часового пояса в стиле POSIX очень легко, так как адекватность аббревиатуры никак не проверяется. Например, команда `SET TIMEZONE TO FOOBAR0` выполнится без ошибки, и система в итоге будет использовать весьма своеобразную аббревиатуру для UTC.

B.6. История единиц измерения времени

Стандарт SQL устанавливает, что «В определении „литерала типа дата-время“, „значения типа дата-время“ ограничены естественными правилами, касающимися дат и времени согласно григорианскому календарю». Следуя стандарту SQL, Postgres Pro подсчитывает даты исключительно в григорианском календаре, включая годы, когда этот календарь ещё не использовался. Это правило известно как *пролептический григорианский календарь*.

Юлианский календарь был введён Юлием Цезарем в 45 г. до н. э. Он широко использовался западной цивилизацией до 1582 года, когда страны начали переходить на григорианский календарь. В юлианском календаре тропический год длится приблизительно 365 1/4 дня = 365,25 дня. Каждые 128 лет накапливается примерно 1 день.

Накапливающаяся погрешность побудила папу Григория XIII реформировать календарь в соответствии с постановлениями Тридентского собора. В григорианском календаре тропический год длится приблизительно 365 + 97 / 400 дней = 365,2425 дней. Таком образом, погрешность в один день тропического года накапливается примерно за 3300 лет.

Приблизительное число 365+97/400 получается из-за того, что из каждых 400 лет 97 високосные. При этом действуют следующие правила:

Каждый год кратный 4 является високосным.
Однако каждый год кратный 100 не является високосным.
Тем не менее каждый год кратный 400 всё же является високосным.

Таким образом, 1700, 1800, 1900, 2100 и 2200 не являются високосными годами. Но 1600, 2000 и 2400 — високосные. А в юлианском календаре високосными считаются все годы, кратные 4.

Папская булла, изданная в феврале 1582 года, предписывала сделать октябрь 1582 года на 10 дней короче, чтобы 15 октября следовало сразу за 4 октября. Это правило соблюдалось в Италии, Польше, Португалии и Испании. Вскоре к ним присоединились и прочие католические страны, но протестантские страны вводили эти изменения неохотно, а страны греческой православной церкви не переходили на новый календарь до начала 20 века. В 1752 г. реформа была проведена в Великобритании и её доминионах (включая территорию сегодняшних Соединённых Штатов Америки). Таким образом, за 2 сентября 1752 года следовало 14 сентября 1752 года. Поэтому в системах Unix программа `cal` выводит следующее:

```
$ cal 9 1752
   September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

Но этот календарь действует только в Великобритании и доминионах. В других местах он является недействительным. Чтобы избежать сложностей и возможной путаницы при отслеживании календарей, которыми фактически пользовались в различных местах в разное время, Postgres Pro применяет правила григорианского календаря ко всем датам, даже если это нарушает историческую достоверность.

Разные календари были составлены в различных частях земного шара, многие из них до григорианской системы. Например, появление китайского календаря относится к 14 веку до н. э. Легенда гласит, что император Хуан-ди изобрёл этот календарь в 2637 г. до н. э. В Китайской Народной Республике григорианский календарь используется для официальных и коммерческих нужд. Китайский календарь используется для определения дат традиционных праздников.

B.7. Юлианские даты

Система *юлианских дат* определяет способ исчисления времени. Она не имеет отношения к юлианскому календарю, несмотря на схожее название. Система юлианских дат была предложена французским ученым Жозефом Жюстом Скалигером (1540-1609) и получила своё название, вероятно, в честь его отца, итальянского ученого Юлия Цезаря Скалигера (1484-1558).

В системе юлианских дат каждый день имеет порядковый номер, начиная с 0 (иногда его называют *номером юлианского дня*). Первый день имеет номер 0 и соответствует 1 января 4713 г. до н. э. по юлианскому календарю или 24 ноября 4714 г. до н. э. по григорианскому календарю. Юлианская дата чаще всего используется в астрономических расчётах для записи ночных наблюдений, и поэтому день длится с полудня до полудня UTC, а не с полуночи до полуночи: Первый юлианский день (JD 0) обозначает 24 часа от полудня UTC 24 ноября 4714 г. до н. э. до полудня UTC 25 ноября 4714 г. до н. э.

Хотя Postgres Pro поддерживает юлианскую дату для записи входных и выходных дат (а также использует юлианские даты для некоторых внутренних вычислений в формате дата-время),

полдень не считается началом суток. Postgres Pro считает юлианский день длящимся от полуночи до полуночи, как и обычный день.

Однако если вам нужно получить астрономическое определение, это можно сделать, произведя вычисления в часовом поясе `UTC+12`. Например,

```
=> SELECT extract(julian from '2021-06-23 7:00:00-04'::timestamptz at time zone 'UTC
+12');
     date_part
--------------------
 2459388.9583333335
(1 row)
=> SELECT extract(julian from '2021-06-23 8:00:00-04'::timestamptz at time zone 'UTC
+12');
 date_part
-----------
   2459389
(1 row)
=> SELECT extract(julian from date '2021-06-23');
 date_part
-----------
   2459389
(1 row)
```

Приложение C. Ключевые слова SQL

В Таблице C.1 перечислены все слова, которые являются ключевыми в стандарте SQL и в Postgres Pro Standard 10.23.1. Общее описание ключевых слов можно найти в Подразделе 4.1.1. (Для экономии места в таблицу включены только две последние версии стандарта SQL и SQL-92 для исторического сравнения. Отличия между ними и другими промежуточными версиями стандарта невелики.)

В SQL есть различие между *зарезервированными* и *незарезервированными* ключевыми словами. Согласно стандарту, действительно ключевыми словами являются только зарезервированные слова; они не могут быть идентификаторами. Незарезервированные ключевые слова имеют особое значение только в определённых контекстах и могут быть идентификаторами в других. Большинство незарезервированных ключевых слов на самом деле представляют имена встроенных таблиц и функций, определённых в SQL. Концепция незарезервированных ключевых слов собственно введена только для того, чтобы показать, что эти слова имеют некоторое предопределённое значение в отдельных контекстах.

В Postgres Pro анализатор SQL сталкивается с дополнительными сложностями. Ему приходится иметь дело с несколькими различными классами элементов языка, начиная с тех, что никогда не могут использоваться как идентификаторы, и заканчивая теми, что не имеют никаких особых отличий от обычных идентификаторов. (Последнее обычно относится к функциям, описанным в SQL.) Даже зарезервированные ключевые слова не полностью зарезервированы в Postgres Pro, а могут использоваться в качестве меток столбцов (например, можно написать `SELECT 55 AS CHECK`, хотя `CHECK` и является зарезервированным ключевым словом).

В Таблице C.1, в столбце Postgres Pro мы даём пометку «не зарезервировано» тем ключевым словам, которые явно известны анализатору запросов, но их можно использовать в качестве имени столбца или таблицы. Некоторые ключевые слова, которые недопустимы в качестве имени функции или типа данных, но в остальном не отличаются от незарезервированных слов, помечены соответственно. (Большинство из этих слов представляют встроенные функции или типы данных со специальным синтаксисом. Функции или типы с таким именем существуют, но пользователь не может их переопределить.) Метка «зарезервировано» даётся тем словам, которые не могут быть именами столбцов или таблиц. Некоторые зарезервированные ключевые слова могут быть именами функций или типов данных; это также отмечается в таблице. Если такой пометки нет, зарезервированное слово допускается только в качестве метки столбца «AS».

Вообще, если вы сталкиваетесь с разнообразными ошибками разбора команд, содержащих в качестве идентификаторов какие-либо из перечисленных ключевых слов, попробуйте для решения проблемы заключить идентификатор в кавычки.

Изучая Таблицу C.1, важно понимать, что отсутствие какого-либо ключевого слова в списке зарезервированных в Postgres Pro не означает, что функциональность, связанная с этим словом, не реализована. И наоборот, присутствие ключевого слова не обязательно говорит о наличии соответствующей функциональности.

Таблица C.1. Ключевые слова SQL

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
`A`		не зарезервировано	не зарезервировано	
`ABORT`	не зарезервировано			
`ABS`		зарезервировано	зарезервировано	
`ABSENT`		не зарезервировано	не зарезервировано	
`ABSOLUTE`	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
ACCESS	не зарезервировано			
ACCORDING		не зарезервировано	не зарезервировано	
ACTION	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
ADA		не зарезервировано	не зарезервировано	не зарезервировано
ADD	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
ADMIN	не зарезервировано	не зарезервировано	не зарезервировано	
AFTER	не зарезервировано	не зарезервировано	не зарезервировано	
AGGREGATE	не зарезервировано			
ALL	зарезервировано	зарезервировано	зарезервировано	зарезервировано
ALLOCATE		зарезервировано	зарезервировано	зарезервировано
ALSO	не зарезервировано			
ALTER	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
ALWAYS	не зарезервировано	не зарезервировано	не зарезервировано	
ANALYSE	зарезервировано			
ANALYZE	зарезервировано			
AND	зарезервировано	зарезервировано	зарезервировано	зарезервировано
ANY	зарезервировано	зарезервировано	зарезервировано	зарезервировано
ARE		зарезервировано	зарезервировано	зарезервировано
ARRAY	зарезервировано	зарезервировано	зарезервировано	
ARRAY_AGG		зарезервировано	зарезервировано	
ARRAY_MAX_ CARDINALITY		зарезервировано		
AS	зарезервировано	зарезервировано	зарезервировано	зарезервировано
ASC	зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
ASENSITIVE		зарезервировано	зарезервировано	
ASSERTION	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
ASSIGNMENT	не зарезервировано	не зарезервировано	не зарезервировано	
ASYMMETRIC	зарезервировано	зарезервировано	зарезервировано	
AT	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
ATOMIC		зарезервировано	зарезервировано	

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
ATTACH	не зарезервировано			
ATTRIBUTE	не зарезервировано	не зарезервировано	не зарезервировано	
ATTRIBUTES		не зарезервировано	не зарезервировано	
AUTHORIZATION	зарезервировано (допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
AVG		зарезервировано	зарезервировано	зарезервировано
BACKWARD	не зарезервировано			
BASE64		не зарезервировано	не зарезервировано	
BEFORE	не зарезервировано	не зарезервировано	не зарезервировано	
BEGIN	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
BEGIN_FRAME		зарезервировано		
BEGIN_PARTITION		зарезервировано		
BERNOULLI		не зарезервировано	не зарезервировано	
BETWEEN	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
BIGINT	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	
BINARY	зарезервировано (допускается функция или тип)	зарезервировано	зарезервировано	
BIT	не зарезервировано (не допускается функция или тип)			зарезервировано
BIT_LENGTH				зарезервировано
BLOB		зарезервировано	зарезервировано	
BLOCKED		не зарезервировано	не зарезервировано	
BOM		не зарезервировано	не зарезервировано	
BOOLEAN	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	
BOTH	зарезервировано	зарезервировано	зарезервировано	зарезервировано

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
BREADTH		не зарезервировано	не зарезервировано	
BY	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
C		не зарезервировано	не зарезервировано	не зарезервировано
CACHE	не зарезервировано			
CALL		зарезервировано	зарезервировано	
CALLED	не зарезервировано	зарезервировано	зарезервировано	
CARDINALITY		зарезервировано	зарезервировано	
CASCADE	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
CASCADED	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
CASE	зарезервировано	зарезервировано	зарезервировано	зарезервировано
CAST	зарезервировано	зарезервировано	зарезервировано	зарезервировано
CATALOG	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
CATALOG_NAME		не зарезервировано	не зарезервировано	не зарезервировано
CEIL		зарезервировано	зарезервировано	
CEILING		зарезервировано	зарезервировано	
CHAIN	не зарезервировано	не зарезервировано	не зарезервировано	
CHAR	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
CHARACTER	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
CHARACTERISTICS	не зарезервировано	не зарезервировано	не зарезервировано	
CHARACTERS		не зарезервировано	не зарезервировано	
CHARACTER_ LENGTH		зарезервировано	зарезервировано	зарезервировано
CHARACTER_SET_ CATALOG		не зарезервировано	не зарезервировано	не зарезервировано
CHARACTER_SET_ NAME		не зарезервировано	не зарезервировано	не зарезервировано
CHARACTER_SET_ SCHEMA		не зарезервировано	не зарезервировано	не зарезервировано

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
CHAR_LENGTH		зарезервировано	зарезервировано	зарезервировано
CHECK	зарезервировано	зарезервировано	зарезервировано	зарезервировано
CHECKPOINT	не зарезервировано			
CLASS	не зарезервировано			
CLASS_ORIGIN		не зарезервировано	не зарезервировано	не зарезервировано
CLOB		зарезервировано	зарезервировано	
CLOSE	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
CLUSTER	не зарезервировано			
COALESCE	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
COBOL		не зарезервировано	не зарезервировано	не зарезервировано
COLLATE	зарезервировано	зарезервировано	зарезервировано	зарезервировано
COLLATION	зарезервировано (допускается функция или тип)	не зарезервировано	не зарезервировано	зарезервировано
COLLATION_ CATALOG		не зарезервировано	не зарезервировано	не зарезервировано
COLLATION_NAME		не зарезервировано	не зарезервировано	не зарезервировано
COLLATION_ SCHEMA		не зарезервировано	не зарезервировано	не зарезервировано
COLLECT		зарезервировано	зарезервировано	
COLUMN	зарезервировано	зарезервировано	зарезервировано	зарезервировано
COLUMNS	не зарезервировано	не зарезервировано	не зарезервировано	
COLUMN_NAME		не зарезервировано	не зарезервировано	не зарезервировано
COMMAND_ FUNCTION		не зарезервировано	не зарезервировано	не зарезервировано
COMMAND_ FUNCTION_CODE		не зарезервировано	не зарезервировано	
COMMENT	не зарезервировано			
COMMENTS	не зарезервировано			
COMMIT	не зарезервировано	зарезервировано	зарезервировано	зарезервировано

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
COMMITTED	не зарезервировано	не зарезервировано	не зарезервировано	не зарезервировано
CONCURRENTLY	зарезервировано (допускается функция или тип)			
CONDITION		зарезервировано	зарезервировано	
CONDITION_NUMBER		не зарезервировано	не зарезервировано	не зарезервировано
CONFIGURATION	не зарезервировано			
CONFLICT	не зарезервировано			
CONNECT		зарезервировано	зарезервировано	зарезервировано
CONNECTION	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
CONNECTION_NAME		не зарезервировано	не зарезервировано	не зарезервировано
CONSTRAINT	зарезервировано	зарезервировано	зарезервировано	зарезервировано
CONSTRAINTS	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
CONSTRAINT_CATALOG		не зарезервировано	не зарезервировано	не зарезервировано
CONSTRAINT_NAME		не зарезервировано	не зарезервировано	не зарезервировано
CONSTRAINT_SCHEMA		не зарезервировано	не зарезервировано	не зарезервировано
CONSTRUCTOR		не зарезервировано	не зарезервировано	
CONTAINS		зарезервировано	не зарезервировано	
CONTENT	не зарезервировано	не зарезервировано	не зарезервировано	
CONTINUE	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
CONTROL		не зарезервировано	не зарезервировано	
CONVERSION	не зарезервировано			
CONVERT		зарезервировано	зарезервировано	зарезервировано
COPY	не зарезервировано			
CORR		зарезервировано	зарезервировано	
CORRESPONDING		зарезервировано	зарезервировано	зарезервировано
COST	не зарезервировано			
COUNT		зарезервировано	зарезервировано	зарезервировано

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
COVAR_POP		зарезервировано	зарезервировано	
COVAR_SAMP		зарезервировано	зарезервировано	
CREATE	зарезервировано	зарезервировано	зарезервировано	зарезервировано
CROSS	зарезервировано (допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
CSV	не зарезервировано			
CUBE	не зарезервировано	зарезервировано	зарезервировано	
CUME_DIST		зарезервировано	зарезервировано	
CURRENT	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
CURRENT_CATALOG	зарезервировано	зарезервировано	зарезервировано	
CURRENT_DATE	зарезервировано	зарезервировано	зарезервировано	зарезервировано
CURRENT_ DEFAULT_ TRANSFORM_ GROUP		зарезервировано	зарезервировано	
CURRENT_PATH		зарезервировано	зарезервировано	
CURRENT_ROLE	зарезервировано	зарезервировано	зарезервировано	
CURRENT_ROW		зарезервировано		
CURRENT_SCHEMA	зарезервировано (допускается функция или тип)	зарезервировано	зарезервировано	
CURRENT_TIME	зарезервировано	зарезервировано	зарезервировано	зарезервировано
CURRENT_ TIMESTAMP	зарезервировано	зарезервировано	зарезервировано	зарезервировано
CURRENT_ TRANSFORM_ GROUP_FOR_ TYPE		зарезервировано	зарезервировано	
CURRENT_USER	зарезервировано	зарезервировано	зарезервировано	зарезервировано
CURSOR	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
CURSOR_NAME		не зарезервировано	не зарезервировано	не зарезервировано
CYCLE	не зарезервировано	зарезервировано	зарезервировано	
DATA	не зарезервировано	не зарезервировано	не зарезервировано	не зарезервировано
DATABASE	не зарезервировано			
DATALINK		зарезервировано	зарезервировано	
DATE		зарезервировано	зарезервировано	зарезервировано

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
DATETIME_ INTERVAL_CODE		не зарезервировано	не зарезервировано	не зарезервировано
DATETIME_ INTERVAL_ PRECISION		не зарезервировано	не зарезервировано	не зарезервировано
DAY	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
DB		не зарезервировано	не зарезервировано	
DEALLOCATE	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
DEC	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
DECIMAL	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
DECLARE	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
DEFAULT	зарезервировано	зарезервировано	зарезервировано	зарезервировано
DEFAULTS	не зарезервировано	не зарезервировано	не зарезервировано	
DEFERRABLE	зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
DEFERRED	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
DEFINED		не зарезервировано	не зарезервировано	
DEFINER	не зарезервировано	не зарезервировано	не зарезервировано	
DEGREE		не зарезервировано	не зарезервировано	
DELETE	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
DELIMITER	не зарезервировано			
DELIMITERS	не зарезервировано			
DENSE_RANK		зарезервировано	зарезервировано	
DEPENDS	не зарезервировано			
DEPTH		не зарезервировано	не зарезервировано	
DEREF		зарезервировано	зарезервировано	

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
DERIVED		не зарезервировано	не зарезервировано	
DESC	зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
DESCRIBE		зарезервировано	зарезервировано	зарезервировано
DESCRIPTOR		не зарезервировано	не зарезервировано	зарезервировано
DETACH	не зарезервировано			
DETERMINISTIC		зарезервировано	зарезервировано	
DIAGNOSTICS		не зарезервировано	не зарезервировано	зарезервировано
DICTIONARY	не зарезервировано			
DISABLE	не зарезервировано			
DISCARD	не зарезервировано			
DISCONNECT		зарезервировано	зарезервировано	зарезервировано
DISPATCH		не зарезервировано	не зарезервировано	
DISTINCT	зарезервировано	зарезервировано	зарезервировано	зарезервировано
DLNEWCOPY		зарезервировано	зарезервировано	
DLPREVIOUSCOPY		зарезервировано	зарезервировано	
DLURLCOMPLETE		зарезервировано	зарезервировано	
DLURLCOMPLETEONLY		зарезервировано	зарезервировано	
DLURLCOMPLETEWRITE		зарезервировано	зарезервировано	
DLURLPATH		зарезервировано	зарезервировано	
DLURLPATHONLY		зарезервировано	зарезервировано	
DLURLPATHWRITE		зарезервировано	зарезервировано	
DLURLSCHEME		зарезервировано	зарезервировано	
DLURLSERVER		зарезервировано	зарезервировано	
DLVALUE		зарезервировано	зарезервировано	
DO	зарезервировано			
DOCUMENT	не зарезервировано	не зарезервировано	не зарезервировано	
DOMAIN	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
DOUBLE	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
DROP	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
DYNAMIC		зарезервировано	зарезервировано	

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
DYNAMIC_ FUNCTION		не зарезервировано	не зарезервировано	не зарезервировано
DYNAMIC_ FUNCTION_CODE		не зарезервировано	не зарезервировано	
EACH	не зарезервировано	зарезервировано	зарезервировано	
ELEMENT		зарезервировано	зарезервировано	
ELSE	зарезервировано	зарезервировано	зарезервировано	зарезервировано
EMPTY		не зарезервировано	не зарезервировано	
ENABLE	не зарезервировано			
ENCODING	не зарезервировано	не зарезервировано	не зарезервировано	
ENCRYPTED	не зарезервировано			
END	зарезервировано	зарезервировано	зарезервировано	зарезервировано
END-EXEC		зарезервировано	зарезервировано	зарезервировано
END_FRAME		зарезервировано		
END_PARTITION		зарезервировано		
ENFORCED		не зарезервировано		
ENUM	не зарезервировано			
EQUALS		зарезервировано	не зарезервировано	
ESCAPE	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
EVENT	не зарезервировано			
EVERY		зарезервировано	зарезервировано	
EXCEPT	зарезервировано	зарезервировано	зарезервировано	зарезервировано
EXCEPTION				зарезервировано
EXCLUDE	не зарезервировано	не зарезервировано	не зарезервировано	
EXCLUDING	не зарезервировано	не зарезервировано	не зарезервировано	
EXCLUSIVE	не зарезервировано			
EXEC		зарезервировано	зарезервировано	зарезервировано
EXECUTE	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
EXISTS	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
EXP		зарезервировано	зарезервировано	
EXPLAIN	не зарезервировано			
EXPRESSION		не зарезервировано		
EXTENSION	не зарезервировано			
EXTERNAL	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
EXTRACT	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
FALSE	зарезервировано	зарезервировано	зарезервировано	зарезервировано
FAMILY	не зарезервировано			
FETCH	зарезервировано	зарезервировано	зарезервировано	зарезервировано
FILE		не зарезервировано	не зарезервировано	
FILTER	не зарезервировано	зарезервировано	зарезервировано	
FINAL		не зарезервировано	не зарезервировано	
FIRST	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
FIRST_VALUE		зарезервировано	зарезервировано	
FLAG		не зарезервировано	не зарезервировано	
FLOAT	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
FLOOR		зарезервировано	зарезервировано	
FOLLOWING	не зарезервировано	не зарезервировано	не зарезервировано	
FOR	зарезервировано	зарезервировано	зарезервировано	зарезервировано
FORCE	не зарезервировано			
FOREIGN	зарезервировано	зарезервировано	зарезервировано	зарезервировано
FORTRAN		не зарезервировано	не зарезервировано	не зарезервировано
FORWARD	не зарезервировано			
FOUND		не зарезервировано	не зарезервировано	зарезервировано
FRAME_ROW		зарезервировано		

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
FREE		зарезервировано	зарезервировано	
FREEZE	зарезервировано (допускается функция или тип)			
FROM	зарезервировано	зарезервировано	зарезервировано	зарезервировано
FS		не зарезервировано	не зарезервировано	
FULL	зарезервировано (допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
FUNCTION	не зарезервировано	зарезервировано	зарезервировано	
FUNCTIONS	не зарезервировано			
FUSION		зарезервировано	зарезервировано	
G		не зарезервировано	не зарезервировано	
GENERAL		не зарезервировано	не зарезервировано	
GENERATED	не зарезервировано	не зарезервировано	не зарезервировано	
GET		зарезервировано	зарезервировано	зарезервировано
GLOBAL	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
GO		не зарезервировано	не зарезервировано	зарезервировано
GOTO		не зарезервировано	не зарезервировано	зарезервировано
GRANT	зарезервировано	зарезервировано	зарезервировано	зарезервировано
GRANTED	не зарезервировано	не зарезервировано	не зарезервировано	
GREATEST	не зарезервировано (не допускается функция или тип)			
GROUP	зарезервировано	зарезервировано	зарезервировано	зарезервировано
GROUPING	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	
GROUPS		зарезервировано		
HANDLER	не зарезервировано			
HAVING	зарезервировано	зарезервировано	зарезервировано	зарезервировано
HEADER	не зарезервировано			

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
HEX		не зарезервировано	не зарезервировано	
HIERARCHY		не зарезервировано	не зарезервировано	
HOLD	не зарезервировано	зарезервировано	зарезервировано	
HOUR	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
ID		не зарезервировано	не зарезервировано	
IDENTITY	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
IF	не зарезервировано			
IGNORE		не зарезервировано	не зарезервировано	
ILIKE	зарезервировано (допускается функция или тип)			
IMMEDIATE	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
IMMEDIATELY		не зарезервировано		
IMMUTABLE	не зарезервировано			
IMPLEMENTATION		не зарезервировано	не зарезервировано	
IMPLICIT	не зарезервировано			
IMPORT	не зарезервировано	зарезервировано	зарезервировано	
IN	зарезервировано	зарезервировано	зарезервировано	зарезервировано
INCLUDING	не зарезервировано	не зарезервировано	не зарезервировано	
INCREMENT	не зарезервировано	не зарезервировано	не зарезервировано	
INDENT		не зарезервировано	не зарезервировано	
INDEX	не зарезервировано			
INDEXES	не зарезервировано			
INDICATOR		зарезервировано	зарезервировано	зарезервировано
INHERIT	не зарезервировано			
INHERITS	не зарезервировано			

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
INITIALLY	зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
INLINE	не зарезервировано			
INNER	зарезервировано (допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
INOUT	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	
INPUT	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
INSENSITIVE	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
INSERT	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
INSTANCE		не зарезервировано	не зарезервировано	
INSTANTIABLE		не зарезервировано	не зарезервировано	
INSTEAD	не зарезервировано	не зарезервировано	не зарезервировано	
INT	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
INTEGER	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
INTEGRITY		не зарезервировано	не зарезервировано	
INTERSECT	зарезервировано	зарезервировано	зарезервировано	зарезервировано
INTERSECTION		зарезервировано	зарезервировано	
INTERVAL	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
INTO	зарезервировано	зарезервировано	зарезервировано	зарезервировано
INVOKER	не зарезервировано	не зарезервировано	не зарезервировано	
IS	зарезервировано (допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
ISNULL	зарезервировано (допускается функция или тип)			

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
ISOLATION	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
JOIN	зарезервировано (допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
K		не зарезервировано	не зарезервировано	
KEY	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
KEY_MEMBER		не зарезервировано	не зарезервировано	
KEY_TYPE		не зарезервировано	не зарезервировано	
LABEL	не зарезервировано			
LAG		зарезервировано	зарезервировано	
LANGUAGE	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
LARGE	не зарезервировано	зарезервировано	зарезервировано	
LAST	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
LAST_VALUE		зарезервировано	зарезервировано	
LATERAL	зарезервировано	зарезервировано	зарезервировано	
LEAD		зарезервировано	зарезервировано	
LEADING	зарезервировано	зарезервировано	зарезервировано	зарезервировано
LEAKPROOF	не зарезервировано			
LEAST	не зарезервировано (не допускается функция или тип)			
LEFT	зарезервировано (допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
LENGTH		не зарезервировано	не зарезервировано	не зарезервировано
LEVEL	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
LIBRARY		не зарезервировано	не зарезервировано	
LIKE	зарезервировано (допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
LIKE_REGEX		зарезервировано	зарезервировано	
LIMIT	зарезервировано	не зарезервировано	не зарезервировано	

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
LINK		не зарезервировано	не зарезервировано	
LISTEN	не зарезервировано			
LN		зарезервировано	зарезервировано	
LOAD	не зарезервировано			
LOCAL	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
LOCALTIME	зарезервировано	зарезервировано	зарезервировано	
LOCALTIMESTAMP	зарезервировано	зарезервировано	зарезервировано	
LOCATION	не зарезервировано	не зарезервировано	не зарезервировано	
LOCATOR		не зарезервировано	не зарезервировано	
LOCK	не зарезервировано			
LOCKED	не зарезервировано			
LOGGED	не зарезервировано			
LOWER		зарезервировано	зарезервировано	зарезервировано
M		не зарезервировано	не зарезервировано	
MAP		не зарезервировано	не зарезервировано	
MAPPING	не зарезервировано	не зарезервировано	не зарезервировано	
MATCH	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
MATCHED		не зарезервировано	не зарезервировано	
MATERIALIZED	не зарезервировано			
MAX		зарезервировано	зарезервировано	зарезервировано
MAXVALUE	не зарезервировано	не зарезервировано	не зарезервировано	
MAX_CARDINALITY			зарезервировано	
MEMBER		зарезервировано	зарезервировано	
MERGE		зарезервировано	зарезервировано	
MESSAGE_LENGTH		не зарезервировано	не зарезервировано	не зарезервировано
MESSAGE_OCTET_ LENGTH		не зарезервировано	не зарезервировано	не зарезервировано
MESSAGE_TEXT		не зарезервировано	не зарезервировано	не зарезервировано

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
METHOD	не зарезервировано	зарезервировано	зарезервировано	
MIN		зарезервировано	зарезервировано	зарезервировано
MINUTE	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
MINVALUE	не зарезервировано	не зарезервировано	не зарезервировано	
MOD		зарезервировано	зарезервировано	
MODE	не зарезервировано			
MODIFIES		зарезервировано	зарезервировано	
MODULE		зарезервировано	зарезервировано	зарезервировано
MONTH	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
MORE		не зарезервировано	не зарезервировано	не зарезервировано
MOVE	не зарезервировано			
MULTISET		зарезервировано	зарезервировано	
MUMPS		не зарезервировано	не зарезервировано	не зарезервировано
NAME	не зарезервировано	не зарезервировано	не зарезервировано	не зарезервировано
NAMES	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
NAMESPACE		не зарезервировано	не зарезервировано	
NATIONAL	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
NATURAL	зарезервировано (допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
NCHAR	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
NCLOB		зарезервировано	зарезервировано	
NESTING		не зарезервировано	не зарезервировано	
NEW	не зарезервировано	зарезервировано	зарезервировано	
NEXT	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
NFC		не зарезервировано	не зарезервировано	

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
NFD		не зарезервировано	не зарезервировано	
NFKC		не зарезервировано	не зарезервировано	
NFKD		не зарезервировано	не зарезервировано	
NIL		не зарезервировано	не зарезервировано	
NO	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
NONE	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	
NORMALIZE		зарезервировано	зарезервировано	
NORMALIZED		не зарезервировано	не зарезервировано	
NOT	зарезервировано	зарезервировано	зарезервировано	зарезервировано
NOTHING	не зарезервировано			
NOTIFY	не зарезервировано			
NOTNULL	зарезервировано (допускается функция или тип)			
NOWAIT	не зарезервировано			
NTH_VALUE		зарезервировано	зарезервировано	
NTILE		зарезервировано	зарезервировано	
NULL	зарезервировано	зарезервировано	зарезервировано	зарезервировано
NULLABLE		не зарезервировано	не зарезервировано	не зарезервировано
NULLIF	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
NULLS	не зарезервировано	не зарезервировано	не зарезервировано	
NUMBER		не зарезервировано	не зарезервировано	не зарезервировано
NUMERIC	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
OBJECT	не зарезервировано	не зарезервировано	не зарезервировано	

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
OCCURRENCES_REGEX		зарезервировано	зарезервировано	
OCTETS		не зарезервировано	не зарезервировано	
OCTET_LENGTH		зарезервировано	зарезервировано	зарезервировано
OF	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
OFF	не зарезервировано	не зарезервировано	не зарезервировано	
OFFSET	зарезервировано	зарезервировано	зарезервировано	
OIDS	не зарезервировано			
OLD	не зарезервировано	зарезервировано	зарезервировано	
ON	зарезервировано	зарезервировано	зарезервировано	зарезервировано
ONLY	зарезервировано	зарезервировано	зарезервировано	зарезервировано
OPEN		зарезервировано	зарезервировано	зарезервировано
OPERATOR	не зарезервировано			
OPTION	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
OPTIONS	не зарезервировано	не зарезервировано	не зарезервировано	
OR	зарезервировано	зарезервировано	зарезервировано	зарезервировано
ORDER	зарезервировано	зарезервировано	зарезервировано	зарезервировано
ORDERING		не зарезервировано	не зарезервировано	
ORDINALITY	не зарезервировано	не зарезервировано	не зарезервировано	
OTHERS		не зарезервировано	не зарезервировано	
OUT	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	
OUTER	зарезервировано (допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
OUTPUT		не зарезервировано	не зарезервировано	зарезервировано
OVER	не зарезервировано	зарезервировано	зарезервировано	
OVERLAPS	зарезервировано (допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
OVERLAY	не зарезервировано (	зарезервировано	зарезервировано	

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
	не допускается функция или тип)			
OVERRIDING	не зарезервировано	не зарезервировано	не зарезервировано	
OWNED	не зарезервировано			
OWNER	не зарезервировано			
P		не зарезервировано	не зарезервировано	
PAD		не зарезервировано	не зарезервировано	зарезервировано
PARALLEL	не зарезервировано			
PARAMETER		зарезервировано	зарезервировано	
PARAMETER_MODE		не зарезервировано	не зарезервировано	
PARAMETER_NAME		не зарезервировано	не зарезервировано	
PARAMETER_ ORDINAL_ POSITION		не зарезервировано	не зарезервировано	
PARAMETER_ SPECIFIC_ CATALOG		не зарезервировано	не зарезервировано	
PARAMETER_ SPECIFIC_NAME		не зарезервировано	не зарезервировано	
PARAMETER_ SPECIFIC_ SCHEMA		не зарезервировано	не зарезервировано	
PARSER	не зарезервировано			
PARTIAL	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
PARTITION	не зарезервировано	зарезервировано	зарезервировано	
PASCAL		не зарезервировано	не зарезервировано	не зарезервировано
PASSING	не зарезервировано	не зарезервировано	не зарезервировано	
PASSTHROUGH		не зарезервировано	не зарезервировано	
PASSWORD	не зарезервировано			
PATH		не зарезервировано	не зарезервировано	
PERCENT		зарезервировано		

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
PERCENTILE_CONT		зарезервировано	зарезервировано	
PERCENTILE_DISC		зарезервировано	зарезервировано	
PERCENT_RANK		зарезервировано	зарезервировано	
PERIOD		зарезервировано		
PERMISSION		не зарезервировано	не зарезервировано	
PLACING	зарезервировано	не зарезервировано	не зарезервировано	
PLANS	не зарезервировано			
PLI		не зарезервировано	не зарезервировано	не зарезервировано
POLICY	не зарезервировано			
PORTION		зарезервировано		
POSITION	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
POSITION_REGEX		зарезервировано	зарезервировано	
POWER		зарезервировано	зарезервировано	
PRECEDES		зарезервировано		
PRECEDING	не зарезервировано	не зарезервировано	не зарезервировано	
PRECISION	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
PREPARE	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
PREPARED	не зарезервировано			
PRESERVE	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
PRIMARY	зарезервировано	зарезервировано	зарезервировано	зарезервировано
PRIOR	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
PRIVILEGES	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
PROCEDURAL	не зарезервировано			
PROCEDURE	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
PROGRAM	не зарезервировано			

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
PUBLIC		не зарезервировано	не зарезервировано	зарезервировано
PUBLICATION	не зарезервировано			
QUOTE	не зарезервировано			
RANGE	не зарезервировано	зарезервировано	зарезервировано	
RANK		зарезервировано	зарезервировано	
READ	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
READS		зарезервировано	зарезервировано	
REAL	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
REASSIGN	не зарезервировано			
RECHECK	не зарезервировано			
RECOVERY		не зарезервировано	не зарезервировано	
RECURSIVE	не зарезервировано	зарезервировано	зарезервировано	
REF	не зарезервировано	зарезервировано	зарезервировано	
REFERENCES	зарезервировано	зарезервировано	зарезервировано	зарезервировано
REFERENCING	не зарезервировано	зарезервировано	зарезервировано	
REFRESH	не зарезервировано			
REGR_AVGX		зарезервировано	зарезервировано	
REGR_AVGY		зарезервировано	зарезервировано	
REGR_COUNT		зарезервировано	зарезервировано	
REGR_INTERCEPT		зарезервировано	зарезервировано	
REGR_R2		зарезервировано	зарезервировано	
REGR_SLOPE		зарезервировано	зарезервировано	
REGR_SXX		зарезервировано	зарезервировано	
REGR_SXY		зарезервировано	зарезервировано	
REGR_SYY		зарезервировано	зарезервировано	
REINDEX	не зарезервировано			
RELATIVE	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
RELEASE	не зарезервировано	зарезервировано	зарезервировано	
RENAME	не зарезервировано			
REPEATABLE	не зарезервировано	не зарезервировано	не зарезервировано	не зарезервировано
REPLACE	не зарезервировано			
REPLICA	не зарезервировано			
REQUIRING		не зарезервировано	не зарезервировано	
RESET	не зарезервировано			
RESPECT		не зарезервировано	не зарезервировано	
RESTART	не зарезервировано	не зарезервировано	не зарезервировано	
RESTORE		не зарезервировано	не зарезервировано	
RESTRICT	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
RESULT		зарезервировано	зарезервировано	
RETURN		зарезервировано	зарезервировано	
RETURNED_CARDINALITY		не зарезервировано	не зарезервировано	
RETURNED_LENGTH		не зарезервировано	не зарезервировано	не зарезервировано
RETURNED_OCTET_LENGTH		не зарезервировано	не зарезервировано	не зарезервировано
RETURNED_SQLSTATE		не зарезервировано	не зарезервировано	не зарезервировано
RETURNING	зарезервировано	не зарезервировано	не зарезервировано	
RETURNS	не зарезервировано	зарезервировано	зарезервировано	
REVOKE	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
RIGHT	зарезервировано (допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
ROLE	не зарезервировано	не зарезервировано	не зарезервировано	
ROLLBACK	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
ROLLUP	не зарезервировано	зарезервировано	зарезервировано	

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
ROUTINE		не зарезервировано	не зарезервировано	
ROUTINE_CATALOG		не зарезервировано	не зарезервировано	
ROUTINE_NAME		не зарезервировано	не зарезервировано	
ROUTINE_SCHEMA		не зарезервировано	не зарезервировано	
ROW	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	
ROWS	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
ROW_COUNT		не зарезервировано	не зарезервировано	не зарезервировано
ROW_NUMBER		зарезервировано	зарезервировано	
RULE	не зарезервировано			
SAVEPOINT	не зарезервировано	зарезервировано	зарезервировано	
SCALE		не зарезервировано	не зарезервировано	не зарезервировано
SCHEMA	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
SCHEMAS	не зарезервировано			
SCHEMA_NAME		не зарезервировано	не зарезервировано	не зарезервировано
SCOPE		зарезервировано	зарезервировано	
SCOPE_CATALOG		не зарезервировано	не зарезервировано	
SCOPE_NAME		не зарезервировано	не зарезервировано	
SCOPE_SCHEMA		не зарезервировано	не зарезервировано	
SCROLL	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
SEARCH	не зарезервировано	зарезервировано	зарезервировано	
SECOND	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
SECTION		не зарезервировано	не зарезервировано	зарезервировано
SECURITY	не зарезервировано	не зарезервировано	не зарезервировано	
SELECT	зарезервировано	зарезервировано	зарезервировано	зарезервировано

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
SELECTIVE		не зарезервировано	не зарезервировано	
SELF		не зарезервировано	не зарезервировано	
SENSITIVE		зарезервировано	зарезервировано	
SEQUENCE	не зарезервировано	не зарезервировано	не зарезервировано	
SEQUENCES	не зарезервировано			
SERIALIZABLE	не зарезервировано	не зарезервировано	не зарезервировано	не зарезервировано
SERVER	не зарезервировано	не зарезервировано	не зарезервировано	
SERVER_NAME		не зарезервировано	не зарезервировано	не зарезервировано
SESSION	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
SESSION_USER	зарезервировано	зарезервировано	зарезервировано	зарезервировано
SET	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
SETOF	не зарезервировано (не допускается функция или тип)			
SETS	не зарезервировано	не зарезервировано	не зарезервировано	
SHARE	не зарезервировано			
SHOW	не зарезервировано			
SIMILAR	зарезервировано (допускается функция или тип)	зарезервировано	зарезервировано	
SIMPLE	не зарезервировано	не зарезервировано	не зарезервировано	
SIZE		не зарезервировано	не зарезервировано	зарезервировано
SKIP	не зарезервировано			
SMALLINT	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
SNAPSHOT	не зарезервировано			
SOME	зарезервировано	зарезервировано	зарезервировано	зарезервировано
SOURCE		не зарезервировано	не зарезервировано	

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
SPACE		не зарезервировано	не зарезервировано	зарезервировано
SPECIFIC		зарезервировано	зарезервировано	
SPECIFICTYPE		зарезервировано	зарезервировано	
SPECIFIC_NAME		не зарезервировано	не зарезервировано	
SQL	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
SQLCODE				зарезервировано
SQLERROR				зарезервировано
SQLEXCEPTION		зарезервировано	зарезервировано	
SQLSTATE		зарезервировано	зарезервировано	зарезервировано
SQLWARNING		зарезервировано	зарезервировано	
SQRT		зарезервировано	зарезервировано	
STABLE	не зарезервировано			
STANDALONE	не зарезервировано	не зарезервировано	не зарезервировано	
START	не зарезервировано	зарезервировано	зарезервировано	
STATE		не зарезервировано	не зарезервировано	
STATEMENT	не зарезервировано	не зарезервировано	не зарезервировано	
STATIC		зарезервировано	зарезервировано	
STATISTICS	не зарезервировано			
STDDEV_POP		зарезервировано	зарезервировано	
STDDEV_SAMP		зарезервировано	зарезервировано	
STDIN	не зарезервировано			
STDOUT	не зарезервировано			
STORAGE	не зарезервировано			
STRICT	не зарезервировано			
STRIP	не зарезервировано	не зарезервировано	не зарезервировано	
STRUCTURE		не зарезервировано	не зарезервировано	
STYLE		не зарезервировано	не зарезервировано	
SUBCLASS_ORIGIN		не зарезервировано	не зарезервировано	не зарезервировано

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
SUBMULTISET		зарезервировано	зарезервировано	
SUBSCRIPTION	не зарезервировано			
SUBSTRING	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
SUBSTRING_REGEX		зарезервировано	зарезервировано	
SUCCEEDS		зарезервировано		
SUM		зарезервировано	зарезервировано	зарезервировано
SYMMETRIC	зарезервировано	зарезервировано	зарезервировано	
SYSID	не зарезервировано			
SYSTEM	не зарезервировано	зарезервировано	зарезервировано	
SYSTEM_TIME		зарезервировано		
SYSTEM_USER		зарезервировано	зарезервировано	зарезервировано
T		не зарезервировано	не зарезервировано	
TABLE	зарезервировано	зарезервировано	зарезервировано	зарезервировано
TABLES	не зарезервировано			
TABLESAMPLE	зарезервировано (допускается функция или тип)	зарезервировано	зарезервировано	
TABLESPACE	не зарезервировано			
TABLE_NAME		не зарезервировано	не зарезервировано	не зарезервировано
TEMP	не зарезервировано			
TEMPLATE	не зарезервировано			
TEMPORARY	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
TEXT	не зарезервировано			
THEN	зарезервировано	зарезервировано	зарезервировано	зарезервировано
TIES		не зарезервировано	не зарезервировано	
TIME	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
TIMESTAMP	не зарезервировано (	зарезервировано	зарезервировано	зарезервировано

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
	не допускается функция или тип)			
TIMEZONE_HOUR		зарезервировано	зарезервировано	зарезервировано
TIMEZONE_MINUTE		зарезервировано	зарезервировано	зарезервировано
TO	зарезервировано	зарезервировано	зарезервировано	зарезервировано
TOKEN		не зарезервировано	не зарезервировано	
TOP_LEVEL_ COUNT		не зарезервировано	не зарезервировано	
TRAILING	зарезервировано	зарезервировано	зарезервировано	зарезервировано
TRANSACTION	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
TRANSACTIONS_ COMMITTED		не зарезервировано	не зарезервировано	
TRANSACTIONS_ ROLLED_BACK		не зарезервировано	не зарезервировано	
TRANSACTION_ ACTIVE		не зарезервировано	не зарезервировано	
TRANSFORM	не зарезервировано	не зарезервировано	не зарезервировано	
TRANSFORMS		не зарезервировано	не зарезервировано	
TRANSLATE		зарезервировано	зарезервировано	зарезервировано
TRANSLATE_REGEX		зарезервировано	зарезервировано	
TRANSLATION		зарезервировано	зарезервировано	зарезервировано
TREAT	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	
TRIGGER	не зарезервировано	зарезервировано	зарезервировано	
TRIGGER_CATALOG		не зарезервировано	не зарезервировано	
TRIGGER_NAME		не зарезервировано	не зарезервировано	
TRIGGER_SCHEMA		не зарезервировано	не зарезервировано	
TRIM	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
TRIM_ARRAY		зарезервировано	зарезервировано	
TRUE	зарезервировано	зарезервировано	зарезервировано	зарезервировано
TRUNCATE	не зарезервировано	зарезервировано	зарезервировано	

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
TRUSTED	не зарезервировано			
TYPE	не зарезервировано	не зарезервировано	не зарезервировано	не зарезервировано
TYPES	не зарезервировано			
UESCAPE		зарезервировано	зарезервировано	
UNBOUNDED	не зарезервировано	не зарезервировано	не зарезервировано	
UNCOMMITTED	не зарезервировано	не зарезервировано	не зарезервировано	не зарезервировано
UNDER		не зарезервировано	не зарезервировано	
UNENCRYPTED	не зарезервировано			
UNION	зарезервировано	зарезервировано	зарезервировано	зарезервировано
UNIQUE	зарезервировано	зарезервировано	зарезервировано	зарезервировано
UNKNOWN	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
UNLINK		не зарезервировано	не зарезервировано	
UNLISTEN	не зарезервировано			
UNLOGGED	не зарезервировано			
UNNAMED		не зарезервировано	не зарезервировано	не зарезервировано
UNNEST		зарезервировано	зарезервировано	
UNTIL	не зарезервировано			
UNTYPED		не зарезервировано	не зарезервировано	
UPDATE	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
UPPER		зарезервировано	зарезервировано	зарезервировано
URI		не зарезервировано	не зарезервировано	
USAGE		не зарезервировано	не зарезервировано	зарезервировано
USER	зарезервировано	зарезервировано	зарезервировано	зарезервировано
USER_DEFINED_ TYPE_CATALOG		не зарезервировано	не зарезервировано	
USER_DEFINED_ TYPE_CODE		не зарезервировано	не зарезервировано	
USER_DEFINED_ TYPE_NAME		не зарезервировано	не зарезервировано	

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
USER_DEFINED_ TYPE_SCHEMA		не зарезервировано	не зарезервировано	
USING	зарезервировано	зарезервировано	зарезервировано	зарезервировано
VACUUM	не зарезервировано			
VALID	не зарезервировано	не зарезервировано	не зарезервировано	
VALIDATE	не зарезервировано			
VALIDATOR	не зарезервировано			
VALUE	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
VALUES	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
VALUE_OF		зарезервировано		
VARBINARY		зарезервировано	зарезервировано	
VARCHAR	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	зарезервировано
VARIADIC	зарезервировано			
VARYING	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
VAR_POP		зарезервировано	зарезервировано	
VAR_SAMP		зарезервировано	зарезервировано	
VERBOSE	зарезервировано (допускается функция или тип)			
VERSION	не зарезервировано	не зарезервировано	не зарезервировано	
VERSIONING		зарезервировано		
VIEW	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
VIEWS	не зарезервировано			
VOLATILE	не зарезервировано			
WHEN	зарезервировано	зарезервировано	зарезервировано	зарезервировано
WHENEVER		зарезервировано	зарезервировано	зарезервировано
WHERE	зарезервировано	зарезервировано	зарезервировано	зарезервировано
WHITESPACE	не зарезервировано	не зарезервировано	не зарезервировано	
WIDTH_BUCKET		зарезервировано	зарезервировано	

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
WINDOW	зарезервировано	зарезервировано	зарезервировано	
WITH	зарезервировано	зарезервировано	зарезервировано	зарезервировано
WITHIN	не зарезервировано	зарезервировано	зарезервировано	
WITHOUT	не зарезервировано	зарезервировано	зарезервировано	
WORK	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
WRAPPER	не зарезервировано	не зарезервировано	не зарезервировано	
WRITE	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано
XML	не зарезервировано	зарезервировано	зарезервировано	
XMLAGG		зарезервировано	зарезервировано	
XMLATTRIBUTES	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	
XMLBINARY		зарезервировано	зарезервировано	
XMLCAST		зарезервировано	зарезервировано	
XMLCOMMENT		зарезервировано	зарезервировано	
XMLCONCAT	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	
XMLDECLARATION		не зарезервировано	не зарезервировано	
XMLDOCUMENT		зарезервировано	зарезервировано	
XMLELEMENT	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	
XMLEXISTS	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	
XMLFOREST	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	
XMLITERATE		зарезервировано	зарезервировано	
XMLNAMESPACES	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	
XMLPARSE	не зарезервировано (	зарезервировано	зарезервировано	

Ключевое слово	Postgres Pro	SQL:2011	SQL:2008	SQL-92
	не допускается функция или тип)			
XMLPI	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	
XMLQUERY		зарезервировано	зарезервировано	
XMLROOT	не зарезервировано (не допускается функция или тип)			
XMLSCHEMA		не зарезервировано	не зарезервировано	
XMLSERIALIZE	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	
XMLTABLE	не зарезервировано (не допускается функция или тип)	зарезервировано	зарезервировано	
XMLTEXT		зарезервировано	зарезервировано	
XMLVALIDATE		зарезервировано	зарезервировано	
YEAR	не зарезервировано	зарезервировано	зарезервировано	зарезервировано
YES	не зарезервировано	не зарезервировано	не зарезервировано	
ZONE	не зарезервировано	не зарезервировано	не зарезервировано	зарезервировано

Приложение D. Соответствие стандарту SQL

В этом разделе в общих чертах отмечается, в какой степени Postgres Pro соответствует текущему стандарту SQL. Следующая информация не является официальным утверждением о соответствии, а представляет только основные аспекты на уровне детализации, достаточно полезном и целесообразном для пользователей.

Формально стандарт SQL называется ISO/IEC 9075 «Database Language SQL» (Язык баз данных SQL). Время от времени выпускается обновлённая версия стандарта; последняя версия стандарта вышла в 2011 г. Эта версия получила обозначение ISO/IEC 9075:2011, или просто SQL:2011. До этого были выпущены версии SQL:2008, SQL:2006, SQL:2003, SQL:1999 и SQL-92. Каждая следующая версия заменяет предыдущую, так что утверждение о совместимости с предыдущими версиями не имеет большой ценности. Разработчики Postgres Pro стремятся обеспечить совместимость с последней официальной версией стандарта, оставаясь при этом в рамках традиционной функциональности и здравого смысла. Postgres Pro реализует большую часть требуемой стандартом функциональности, хотя иногда с немного другими функциями или синтаксисом. Можно ожидать, что со временем степень совместимости будет увеличиваться.

SQL-92 определяет три уровня функциональной совместимости: начальный (Entry), промежуточный (Intermediate) и полный (Full). Большинство СУБД заявляют о совместимости со стандартом SQL только на начальном уровне, так как полный набор возможностей на промежуточном и полном уровнях либо слишком велик, либо конфликтует с ранее принятым поведением.

Начиная с SQL:1999, вместо трёх чрезмерно пространных уровней SQL-92 в стандарте SQL определено множество отдельных функциональных возможностей. Большое его подмножество представляет «Основную» функциональность, которую должны обеспечивать все совместимые с SQL реализации. Поддержка остальных возможностей не является обязательной. Некоторые необязательные возможности группируются вместе, образуя «пакеты», о поддержке которых могут заявлять реализации SQL, таким образом, декларируя совместимость с определённой группой возможностей.

Описание стандарта, начиная с версии SQL:2003, также разделяется на несколько частей. Каждая такая часть имеет короткое имя и номер. Заметьте, что нумерация этих частей непоследовательная.

- ISO/IEC 9075-1 Структура (SQL/Framework)
- ISO/IEC 9075-2 Основа (SQL/Foundation)
- ISO/IEC 9075-3 Интерфейс уровня вызовов (SQL/CLI)
- ISO/IEC 9075-4 Модули постоянного хранения (SQL/PSM)
- ISO/IEC 9075-9 Управление внешними данными (SQL/MED)
- ISO/IEC 9075-10 Привязки объектных языков (SQL/OLB)
- ISO/IEC 9075-11 Схемы информации и определений (SQL/Schemata)
- ISO/IEC 9075-13 Программы и типы, использующие язык Java (SQL/JRT)
- ISO/IEC 9075-14 Спецификации, связанные с XML (SQL/XML)

Ядро Postgres Pro реализует части 1, 2, 9, 11 и 14. Часть 3 реализуется драйвером ODBC, а часть 13 — подключаемым расширением PL/Java, но точное соответствие этих компонентов стандарту на данный момент не проверено. Части 4 и 10 в Postgres Pro в настоящее время не реализованы.

Postgres Pro поддерживает почти все основные возможности стандарта SQL:2011. Из 179 обязательных возможностей, которые требуются для полного соответствия «Основной» функциональности, Postgres Pro обеспечивает совместимость как минимум для 160. Кроме того, он реализует длинный список необязательных возможностей. Следует отметить, что на время написания этой документации ни одна существующая СУБД не заявила о полном соответствии «Основной» функциональности SQL:2011.

В следующих двух разделах мы представляем список возможностей, которые поддерживает Postgres Pro, и список возможностей, определённых в SQL:2011, которые ещё не поддерживаются в Postgres Pro. Оба эти списка носят приблизительный характер: какая-то возможность, отмеченная как поддерживаемая, может отличаться от стандарта в деталях, и напротив, для какой-то неподдерживаемой возможности могут быть реализованы ключевые компоненты. Наиболее точная информация о том, что работает, а что нет, содержится в основной документации.

Примечание

Коды возможностей, содержащие знак минус, обозначают подчинённые возможности. При этом, если какая-либо одна подчинённая возможность не поддерживается, основная возможность так же не будет поддерживаться, даже если реализованы все остальные на подуровне.

D.1. Поддерживаемые возможности

Идентификатор	Пакет	Описание	Комментарий
B012		Встроенный C	
B021		Непосредственный SQL	
E011	Основа	Числовые типы данных	
E011-01	Основа	Типы данных INTEGER и SMALLINT	
E011-02	Основа	Типы данных REAL, DOUBLE PRECISION и FLOAT	
E011-03	Основа	Типы данных DECIMAL и NUMERIC	
E011-04	Основа	Арифметические операторы	
E011-05	Основа	Числовые сравнения	
E011-06	Основа	Неявные преобразования между числовыми типами данных	
E021	Основа	Символьные типы данных	
E021-01	Основа	Тип данных CHARACTER	
E021-02	Основа	Тип данных CHARACTER VARYING	
E021-03	Основа	Символьные строки	

Идентификатор	Пакет	Описание	Комментарий
E021-04	Основа	Функция CHARACTER_LENGTH	убирает завершающие пробелы из значений CHARACTER перед подсчётом символов
E021-05	Основа	Функция OCTET_LENGTH	
E021-06	Основа	Функция SUBSTRING	
E021-07	Основа	Конкатенация символьных строк	
E021-08	Основа	Функции UPPER и LOWER	
E021-09	Основа	Функция TRIM	
E021-10	Основа	Неявные преобразования между типами символьных строк	
E021-11	Основа	Функция POSITION	
E021-12	Основа	Сравнения символов	
E031	Основа	Идентификаторы	
E031-01	Основа	Идентификаторы с разделителями	
E031-02	Основа	Идентификаторы в нижнем регистре	
E031-03	Основа	Завершающее подчёркивание	
E051	Основа	Базовое определение запросов	
E051-01	Основа	SELECT DISTINCT	
E051-02	Основа	Предложение GROUP BY	
E051-04	Основа	GROUP BY может содержать столбцы не из <списка выборки>	
E051-05	Основа	Элементы списка выборки могут переименовываться	
E051-06	Основа	Предложение HAVING	
E051-07	Основа	Дополнение * в списке выборки	
E051-08	Основа	Корреляционные имена в предложении FROM	
E051-09	Основа	Переименование столбцов в предложении FROM	
E061	Основа	Базовые предикаты и условия поиска	

Идентификатор	Пакет	Описание	Комментарий
E061-01	Основа	Предикат сравнения	
E061-02	Основа	Предикат BETWEEN	
E061-03	Основа	Предикат IN со списком значений	
E061-04	Основа	Предикат LIKE	
E061-05	Основа	Предложение ESCAPE в предикате LIKE	
E061-06	Основа	Предикат NULL	
E061-07	Основа	Предикаты количественного сравнения	
E061-08	Основа	Предикат EXISTS	
E061-09	Основа	Подзапросы в предикате сравнения	
E061-11	Основа	Подзапросы в предикате IN	
E061-12	Основа	Подзапросы в предикате количественного сравнения	
E061-13	Основа	Коррелирующие подзапросы	
E061-14	Основа	Условие поиска	
E071	Основа	Простые выражения с запросами	
E071-01	Основа	Табличный оператор UNION DISTINCT	
E071-02	Основа	Табличный оператор UNION ALL	
E071-03	Основа	Табличный оператор EXCEPT DISTINCT	
E071-05	Основа	Столбцы, объединяемые табличными операторами, могут иметь разные типы данных	
E071-06	Основа	Табличные операторы в подзапросах	
E081	Основа	Основные права доступа	
E081-01	Основа	Право на SELECT	
E081-02	Основа	Право на DELETE	
E081-03	Основа	Право на INSERT на уровне таблицы	
E081-04	Основа	Право на UPDATE на уровне таблицы	

Идентификатор	Пакет	Описание	Комментарий
E081-05	Основа	Право на UPDATE на уровне столбцов	
E081-06	Основа	Право REFERENCES на уровне таблицы	
E081-07	Основа	Право REFERENCES на уровне столбцов	
E081-08	Основа	Предложение WITH GRANT OPTION	
E081-09	Основа	Право USAGE	
E081-10	Основа	Право на EXECUTE	
E091	Основа	Функции множеств	
E091-01	Основа	AVG	
E091-02	Основа	COUNT	
E091-03	Основа	MAX	
E091-04	Основа	MIN	
E091-05	Основа	SUM	
E091-06	Основа	Дополнение ALL	
E091-07	Основа	Дополнение DISTINCT	
E101	Основа	Базовая обработка данных	
E101-01	Основа	Оператор INSERT	
E101-03	Основа	Оператор UPDATE с критерием отбора	
E101-04	Основа	Оператор DELETE с критерием отбора	
E111	Основа	Оператор SELECT, возвращающий одну строку	
E121	Основа	Базовая поддержка курсоров	
E121-01	Основа	DECLARE CURSOR	
E121-02	Основа	Столбцы ORDER BY, отсутствующие в списке выборки	
E121-03	Основа	Выражения значений в предложении ORDER BY	
E121-04	Основа	Оператор OPEN	
E121-06	Основа	Оператор UPDATE с позиционированием	
E121-07	Основа	Оператор DELETE с позиционированием	
E121-08	Основа	Оператор CLOSE	
E121-10	Основа	Оператор FETCH с неявным NEXT	

Идентификатор	Пакет	Описание	Комментарий
E121-17	Основа	Курсоры WITH HOLD	
E131	Основа	Поддержка NULL (NULL вместо значений)	
E141	Основа	Основные ограничения целостности	
E141-01	Основа	Ограничения NOT NULL	
E141-02	Основа	Ограничения UNIQUE столбцов NOT NULL	
E141-03	Основа	Ограничения PRIMARY KEY	
E141-04	Основа	Базовое ограничение FOREIGN KEY без действия (NO ACTION) по умолчанию и для операций удаления со ссылками, и для операций изменения со ссылками	
E141-06	Основа	Ограничения CHECK	
E141-07	Основа	Значения столбцов по умолчанию	
E141-08	Основа	NOT NULL распространяется на PRIMARY KEY	
E141-10	Основа	Имена во внешнем ключе могут указываться в любом порядке	
E151	Основа	Поддержка транзакций	
E151-01	Основа	Оператор COMMIT	
E151-02	Основа	Оператор ROLLBACK	
E152	Основа	Базовый оператор SET TRANSACTION	
E152-01	Основа	Оператор SET TRANSACTION: предложение ISOLATION LEVEL SERIALIZABLE	
E152-02	Основа	Оператор SET TRANSACTION: предложения READ ONLY и READ WRITE	
E153	Основа	Запросы, изменяющие данные, с подзапросами	
E161	Основа	Комментарии SQL, начинающиеся с двух минусов	

Идентификатор	**Пакет**	**Описание**	**Комментарий**
E171	Основа	Поддержка SQLSTATE	
F021	Основа	Основная информационная схема	
F021-01	Основа	Представление COLUMNS	
F021-02	Основа	Представление TABLES	
F021-03	Основа	Представление VIEWS	
F021-04	Основа	Представление TABLE_CONSTRAINTS	
F021-05	Основа	Представление REFERENTIAL_CONSTRAINTS	
F021-06	Основа	Представление CHECK_CONSTRAINTS	
F031	Основа	Базовые манипуляции со схемой	
F031-01	Основа	Оператор CREATE TABLE создаёт хранимые основные таблицы	
F031-02	Основа	Представление CREATE VIEW	
F031-03	Основа	Оператор GRANT	
F031-04	Основа	Оператор ALTER TABLE: предложение ADD COLUMN	
F031-13	Основа	Оператор DROP TABLE: предложение RESTRICT	
F031-16	Основа	Оператор DROP VIEW: предложение RESTRICT	
F031-19	Основа	Оператор REVOKE: предложение RESTRICT	
F032		Каскадное удаление (CASCADE)	
F033		Оператор ALTER TABLE: предложение DROP COLUMN	
F034		Расширенный оператор REVOKE	
F034-01		Оператор REVOKE может выполняться не только владельцем объекта схемы	

Идентификатор	Пакет	Описание	Комментарий
F034-02		Оператор REVOKE: предложение GRANT OPTION FOR	
F034-03		Оператор REVOKE отзывает право, данное субъекту с указанием WITH GRANT OPTION	
F041	Основа	Базовое соединение таблиц	
F041-01	Основа	Внутреннее соединение (но не обязательно с ключевым словом INNER)	
F041-02	Основа	Ключевое слово INNER	
F041-03	Основа	LEFT OUTER JOIN	
F041-04	Основа	RIGHT OUTER JOIN	
F041-05	Основа	Внешние соединения могут быть вложенными	
F041-07	Основа	Внутренняя таблица с левой или правой стороны внешнего соединения может также участвовать во внутреннем соединении	
F041-08	Основа	Поддерживаются все операторы сравнения (а не только =)	
F051	Основа	Базовая поддержка даты и времени	
F051-01	Основа	Тип данных DATE (включая поддержку строк DATE)	
F051-02	Основа	Тип данных TIME (включая поддержку строк TIME) с точностью до секунд как минимум с 0 знаков после запятой	
F051-03	Основа	Тип данных TIMESTAMP (включая поддержку строк TIMESTAMP) с точностью до секунд как минимум с 0 и 6 знаками после запятой	
F051-04	Основа	Предикаты сравнения с типами данных DATE, TIME и TIMESTAMP	

Идентификатор	Пакет	Описание	Комментарий
F051-05	Основа	Явное приведение (CAST) между типами даты/времени и типами символьных строк	
F051-06	Основа	CURRENT_DATE	
F051-07	Основа	LOCALTIME	
F051-08	Основа	LOCALTIMESTAMP	
F052	Расширенные средства работы с датами/временем	Арифметика с интервалами и датами/временем	
F053		Предикат OVERLAPS	
F081	Основа	UNION и EXCEPT в представлениях	
F111		Уровни изоляции, отличные от SERIALIZABLE	
F111-01		Уровень изоляции READ UNCOMMITTED	
F111-02		Уровень изоляции READ COMMITTED	
F111-03		Уровень изоляции REPEATABLE READ	
F131	Основа	Операции группировки	
F131-01	Основа	Предложения WHERE, GROUP BY и HAVING, поддерживаемые в запросах со сгруппированными представлениями	
F131-02	Основа	Поддержка нескольких таблиц в запросах со сгруппированными представлениями	
F131-03	Основа	Поддержка функций множеств в запросах со сгруппированными представлениями	
F131-04	Основа	Подзапросы с предложениями GROUP BY и HAVING и сгруппированные представления	
F131-05	Основа	SELECT, возвращающий одну строку, с предложениями GROUP BY и HAVING и сгруппированными представлениями	

Идентификатор	Пакет	Описание	Комментарий
F171		Несколько схем для одного пользователя	
F191	Расширенное управление целостностью	Действия при удалении со ссылками	
F200		Оператор TRUNCATE TABLE	
F201	Основа	Функция CAST	
F202		TRUNCATE TABLE: возможность перезапуска идентифицирующего столбца	
F221	Основа	Явные значения по умолчанию	
F222		Оператор INSERT: предложение DEFAULT VALUES	
F231		Таблицы прав	
F231-01		Представление TABLE_PRIVILEGES	
F231-02		Представление COLUMN_PRIVILEGES	
F231-03		Представление USAGE_PRIVILEGES	
F251		Поддержка доменов	
F261	Основа	Выражение CASE	
F261-01	Основа	Простой оператор CASE	
F261-02	Основа	Оператор CASE с условиями	
F261-03	Основа	NULLIF	
F261-04	Основа	COALESCE	
F262		Расширенные выражения CASE	
F271		Составные строки символов	
F281		Улучшенный оператор LIKE	
F302		Табличный оператор INTERSECT	
F302-01		Табличный оператор INTERSECT DISTINCT	
F302-02		Табличный оператор INTERSECT ALL	
F304		Табличный оператор EXCEPT ALL	

Идентификатор	Пакет	Описание	Комментарий
F311-01	Основа	CREATE SCHEMA	
F311-02	Основа	CREATE TABLE для хранимых основных таблиц	
F311-03	Основа	CREATE VIEW	
F311-04	Основа	CREATE VIEW: WITH CHECK OPTION	
F311-05	Основа	Оператор GRANT	
F321		Авторизация пользователей	
F361		Поддержка подпрограмм	
F381		Расширенные манипуляции со схемой	
F381-01		Оператор ALTER TABLE: предложение ALTER COLUMN	
F381-02		Оператор ALTER TABLE: предложение ADD CONSTRAINT	
F381-03		Оператор ALTER TABLE: предложение DROP CONSTRAINT	
F382		Изменение типа данных столбцов	
F383		Предложение, устанавливающее NOT NULL для столбца	
F384		Предложение удаления свойства идентифицирующего столбца	
F386		Предложение установления генерирования значений идентифицирующего столбца	
F391		Длинные идентификаторы	
F392		Спецсимволы Unicode в идентификаторах	
F393		Спецсимволы Unicode в текстовых строках	
F401		Расширенное соединение таблиц	
F401-01		NATURAL JOIN	
F401-02		FULL OUTER JOIN	

Идентификатор	**Пакет**	**Описание**	**Комментарий**
F401-04		CROSS JOIN	
F402		Соединения по именам столбцов для больших объектов, массивов и мультимножеств	
F411	Расширенные средства работы с датами/временем	Указание часового пояса	отличия в интерпретации строкового представления
F421		Национальные символы	
F431		Прокручиваемые курсоры только для чтения	
F431-01		FETCH с явным NEXT	
F431-02		FETCH FIRST	
F431-03		FETCH LAST	
F431-04		FETCH PRIOR	
F431-05		FETCH ABSOLUTE	
F431-06		FETCH RELATIVE	
F441		Расширенная поддержка функций множеств	
F442		Смешанные ссылки на столбцы в функциях множеств	
F471	Основа	Скалярные значения подзапросов	
F481	Основа	Расширенный предикат NULL	
F491	Расширенное управление целостностью	Управление ограничениями	
F501	Основа	Представления возможностей и совместимости	
F501-01	Основа	Представление SQL_FEATURES	
F501-02	Основа	Представление SQL_SIZING	
F501-03	Основа	Представление SQL_LANGUAGES	
F502		Таблицы расширенной документации	
F502-01		Представление SQL_SIZING_PROFILES	

Идентификатор	Пакет	Описание	Комментарий
F502-02		Представление SQL_IMPLEMENTATION_INFO	
F502-03		Представление SQL_PACKAGES	
F531		Временные таблицы	
F555	Расширенные средства работы с датами/временем	Дополнительная точность в секундах	
F561		Полные выражения значений	
F571		Проверки значений истинности	
F591		Производные таблицы	
F611		Типы данных для индикаторов	
F641		Конструкторы строк и таблиц	
F651		Дополнения имён каталогов	
F661		Простые таблицы	
F672		Ограничения-проверки с текущим временем	
F690		Поддержка правил сортировки	но без поддержки наборов символов
F692		Расширенная поддержка правил сортировки	
F701	Расширенное управление целостностью	Действия при обновлении со ссылками	
F711		ALTER для домена	
F731		Права на INSERT для столбцов	
F751		Усовершенствования CHECK для представлений	
F761		Управление сеансом	
F762		CURRENT_CATALOG	
F763		CURRENT_SCHEMA	
F771		Управление соединением	
F781		Самоссылающиеся операции	
F791		Нечувствительные курсоры	

Идентификатор	Пакет	Описание	Комментарий
F801		Полные функции множеств	
F850		<Предложение order by> на верхнем уровне в <выражении запроса>	
F851		<Предложение order by> в подзапросах	
F852		<Предложение order by> на верхнем уровне в представлениях	
F855		Вложенное <предложение order by> в <выражении запроса>	
F856		Вложенное <предложение fetch first> в <предложении запроса>	
F857		<Предложение fetch first> на верхнем уровне в <выражении запроса>	
F858		<Предложение fetch first> в подзапросах	
F859		<Предложение fetch first> на верхнем уровне в представлениях	
F860		<Указание числа строк> в <предложении fetch first>	
F861		<Предложение offset для результата> на верхнем уровне в <выражении запроса>	
F862		<Предложение offset для результата> в подзапросах	
F863		Вложенное <предложение offset для результата> в <выражении запроса>	
F864		<Предложение offset для результата> на верхнем уровне в представлениях	
F865		<Указание числа строк> с <предложением offset для результата>	

Идентификатор	Пакет	Описание	Комментарий
S071	Расширенная поддержка объектов	SQL-пути при разрешении имён функций и типов	
S092		Массивы пользовательских типов	
S095		Конструкторы массива из запроса	
S096		Необязательное указание границ массива	
S098		ARRAY_AGG	
S111	Расширенная поддержка объектов	ONLY в выражениях запросов	
S201		Вызываемые из SQL подпрограммы, работающие с массивами	
S201-01		Массивы в параметрах	
S201-02		Массивы в качестве типа результата функций	
S211	Расширенная поддержка объектов	Пользовательские функции приведений	
S301		Расширенный UNNEST	
T031		Тип данных BOOLEAN	
T071		Тип данных BIGINT	
T121		WITH (без RECURSIVE) в выражении запроса	
T122		WITH (с RECURSIVE) в подзапросе	
T131		Рекурсивный запрос	
T132		Рекурсивный запрос в подзапросе	
T141		Предикат SIMILAR	
T151		Предикат DISTINCT	
T152		Предикат DISTINCT с отрицанием	
T171		Предложение LIKE в определении таблицы	
T172		Предложение подзапроса AS в определении таблицы	
T173		Расширенное предложение LIKE в определении таблицы	

Идентификатор	Пакет	Описание	Комментарий
T174		Идентифицирующие столбцы	
T177		Поддержка генераторов последовательностей: возможность простого перезапуска	
T178		Идентифицирующие столбцы: возможность простого перезапуска	
T191	Расширенное управление целостностью	Действие RESTRICT при нарушении ссылок	
T201	Расширенное управление целостностью	Сравнимые типы данных для ссылочных ограничений	
T211-01	Активная база данных, улучшенное управление целостностью	Триггеры, активируемые при UPDATE, INSERT или DELETE в одной базовой таблице	
T211-02	Активная база данных, улучшенное управление целостностью	Триггеры BEFORE	
T211-03	Активная база данных, улучшенное управление целостностью	Триггеры AFTER	
T211-04	Активная база данных, улучшенное управление целостностью	Триггеры FOR EACH ROW	
T211-05	Активная база данных, улучшенное управление целостностью	Возможность задать условие поиска, которое должно быть истинным перед вызовом триггера	
T211-07	Активная база данных, улучшенное управление целостностью	Право TRIGGER	
T212	Расширенное управление целостностью	Расширенные возможности триггеров	
T213		Триггеры INSTEAD OF	
T231		Чувствительные курсоры	
T241		Оператор START TRANSACTION	

Идентификатор	Пакет	Описание	Комментарий
T271		Точки сохранения	
T281		Право SELECT на уровне столбцов	
T285		Улучшения имён производных столбцов	
T312		Функция OVERLAY	
T321-01	Основа	Пользовательские функции без перегрузки	
T321-03	Основа	Вызов функций	
T321-06	Основа	Представление ROUTINES	
T321-07	Основа	Представление PARAMETERS	
T323		Явное управление безопасностью внешних подпрограмм	
T325		Дополненные указания параметров SQL	
T331		Базовые роли	
T341		Перегрузка вызываемых из SQL функций и процедур	
T351		Блочные комментарии SQL (комментарии /*...*/)	
T431	OLAP	Расширенные возможности группирования	
T432		Вложения и конкатенация GROUPING SETS	
T433		Функция GROUPING с несколькими аргументами	
T441		Функции ABS и MOD	
T461		Симметричный предикат BETWEEN	
T491		Производная таблица LATERAL	
T501		Улучшенный предикат EXISTS	
T551		Необязательные ключевые слова, подразумеваемые синтаксисом по умолчанию	

Идентификатор	Пакет	Описание	Комментарий
T581		Функция подстроки по регулярному выражению	
T591		Ограничения UNIQUE для столбцов, принимающих NULL	
T611	OLAP	Элементарные операции OLAP	
T613		Получение выборки	
T614		Функция NTILE	
T615		Функции LEAD и LAG	
T617		Функции FIRST_VALUE и LAST_VALUE	
T621		Дополнительные численные функции	
T631	Основа	Предикат IN с одним элементом списка	
T651		Операторы модификации схемы SQL в SQL-подпрограммах	
T655		Циклически зависимые подпрограммы	
X010		Тип XML	
X011		Массивы типа XML	
X014		Атрибуты типа XML	
X016		Хранимые значения XML	
X020		XMLConcat	
X031		XMLElement	
X032		XMLForest	
X034		XMLAgg	
X035		XMLAgg: параметр ORDER BY	
X036		XMLComment	
X037		XMLPI	
X040		Базовое отображение таблиц	
X041		Базовое отображение таблиц: значения NULL отсутствуют	
X042		Базовое отображение таблиц: NULL в виде nil	
X043		Базовое отображение таблиц: таблица в виде леса элементов	

Идентификатор	Пакет	Описание	Комментарий
X044		Базовое отображение таблиц: таблица в виде элемента	
X045		Базовое отображение таблиц: с целевым пространством имён	
X046		Базовое отображение таблиц: отображение данных	
X047		Базовое отображение таблиц: отображение метаданных	
X048		Базовое отображение таблиц: кодирование двоичных строк в base64	
X049		Базовое отображение таблиц: кодирование двоичных строк в шестнадцатеричном виде	
X050		Расширенное отображение таблиц	
X051		Расширенное отображение таблиц: значения NULL отсутствуют	
X052		Расширенное отображение таблиц: NULL в виде nil	
X053		Расширенное отображение таблиц: таблица в виде леса элементов	
X054		Расширенное отображение таблиц: таблица в виде элемента	
X055		Расширенное отображение таблиц: с целевым пространством имён	
X056		Расширенное отображение таблиц: отображение данных	
X057		Расширенное отображение таблиц: отображение метаданных	
X058		Расширенное отображение таблиц:	

Идентификатор	Пакет	Описание	Комментарий
		кодирование двоичных строк в base64	
X059		Расширенное отображение таблиц: кодирование двоичных строк в шестнадцатеричном виде	
X060		XMLParse: ввод символьных строк и вариант CONTENT	
X061		XMLParse: ввод символьных строк и вариант DOCUMENT	
X070		XMLSerialize: сериализация символьных строк и вариант CONTENT	
X071		XMLSerialize: сериализация символьных строк и вариант DOCUMENT	
X072		XMLSerialize: сериализация символьных строк	
X090		Предикат XML-документа	
X120		XML в параметрах SQL-подпрограмм	
X121		XML в параметрах внешних подпрограмм	
X222		Механизм передачи XML BY REF	
X301		XMLTable: указание списка производных столбцов	
X302		XMLTable: указание столбца нумерации	
X303		XMLTable: указание значения столбца по умолчанию	
X304		XMLTable: передача контекста	требуется XML DOCUMENT
X400		Сопоставление имён и идентификаторов	
X410		Изменение типа данных столбца: поддержка типа XML	

D.2. Неподдерживаемые возможности

Следующие возможности, описанные в SQL:2011, не реализованы в этом выпуске Postgres Pro. В некоторых случаях они заменяются равнозначной функциональностью

Идентификатор	Пакет	Описание	Комментарий
B011		Встроенный язык Ada	
B013		Встроенный язык COBOL	
B014		Встроенный язык Fortran	
B015		Встроенный язык MUMPS	
B016		Встроенный язык Pascal	
B017		Встроенный язык PL/I	
B031		Базовый динамический SQL	
B032		Расширенный динамический SQL	
B032-01		<Оператор describe input>	
B033		Нетипизированные аргументы функции, вызываемой из SQL	
B034		Динамическое указание атрибутов курсора	
B035		Нерасширенные имена дескрипторов	
B041		Расширения встроенных объявлений исключений SQL	
B051		Расширенные права для выполнения	
B111		Язык модулей — Ada	
B112		Язык модулей — C	
B113		Язык модулей — COBOL	
B114		Язык модулей — Fortran	
B115		Язык модулей — MUMPS	
B116		Язык модулей — Pascal	
B117		Язык модулей — PL/I	
B121		Язык подпрограмм — Ada	
B122		Язык подпрограмм — C	

Идентификатор	Пакет	Описание	Комментарий
B123		Язык подпрограмм — COBOL	
B124		Язык подпрограмм — Fortran	
B125		Язык подпрограмм — MUMPS	
B126		Язык подпрограмм — Pascal	
B127		Язык подпрограмм — PL/I	
B128		Язык подпрограмм — SQL	
B211		Язык модулей — Ada: поддержка VARCHAR и NUMERIC	
B221		Язык подпрограмм — Ada: поддержка VARCHAR и NUMERIC	
E182	Основа	Язык модулей	
F054		TIMESTAMP в списке приоритетов типа DATE	
F121		Базовое управление диагностикой	
F121-01		Оператор GET DIAGNOSTICS	
F121-02		Оператор SET TRANSACTION: предложение DIAGNOSTICS SIZE	
F122		Расширенное управление диагностикой	
F123		Вся диагностика	
F181	Основа	Поддержка множества модулей	
F263		Разделённые запятыми предикаты в простом выражении CASE	
F291		Предикат UNIQUE	
F301		CORRESPONDING в выражениях запросов	
F311	Основа	Оператор определения схемы	
F312		Оператор MERGE	возможная альтернатива — INSERT ... ON CONFLICT DO UPDATE

Идентификатор	**Пакет**	**Описание**	**Комментарий**
F313		Расширенный оператор MERGE	
F314		Оператор MERGE с ветвью DELETE	
F341		Таблицы использования	без таблиц ROUTINE_*_USAGE
F385		Предложение удаления выражения, генерирующего значения столбца	
F394		Необязательное указание нормальной формы	
F403		Секционированные соединённые таблицы	
F451		Определение набора символов	
F461		Именованные наборы символов	
F492		Необязательное указание соблюдения ограничения таблицы	
F521	Расширенное управление целостностью	Утверждения	
F671	Расширенное управление целостностью	Подзапросы в CHECK	намеренно опущено
F693		Правила сортировки символов для SQL-сеансов и клиентских модулей	
F695		Поддержка перекодировки	
F696		Дополнительная документация по перекодировке	
F721		Откладываемые ограничения	только сторонние и уникальные ключи
F741		Типы ссылочных совпадений MATCH	пока без частичного совпадения
F812	Основа	Базовое флагирование	
F813		Расширенное флагирование	
F821		Ссылки на локальные таблицы	
F831		Полное изменение курсора	

Идентификатор	Пакет	Описание	Комментарий
F831-01		Изменяемые прокручиваемые курсоры	
F831-02		Изменяемые упорядоченные курсоры	
F841		Предикат LIKE_REGEX	
F842		Функция OCCURRENCES_REGEX	
F843		Функция POSITION_REGEX	
F844		Функция SUBSTRING_REGEX	
F845		Функция TRANSLATE_REGEX	
F846		Поддержка октетов в операторах регулярных выражений	
F847		Неконстантные регулярные выражения	
F866		Предложение FETCH FIRST: параметр PERCENT	
F867		Предложение FETCH FIRST: параметр WITH TIES	
S011	Основа	Отдельные типы данных	
S011-01	Основа	Представление USER_DEFINED_TYPES	
S023	Базовая поддержка объектов	Базовые структурированные типы	
S024	Расширенная поддержка объектов	Расширенные структурированные типы	
S025		Окончательные структурированные типы	
S026		Самоссылающиеся структурированные типы	
S027		Создание метода по заданному имени метода	
S028		Произвольный порядок параметров UDT	

Идентификатор	Пакет	Описание	Комментарий
S041	Базовая поддержка объектов	Базовые ссылочные типы	
S043	Расширенная поддержка объектов	Расширенные ссылочные типы	
S051	Базовая поддержка объектов	Создание таблицы из типа	частично поддерживается
S081	Расширенная поддержка объектов	Подтаблицы	
S091		Базовая поддержка массивов	частично поддерживается
S091-01		Массивы встроенных типов данных	
S091-02		Массивы отдельных типов	
S091-03		Выражения с массивами	
S094		Массивы ссылочных типов	
S097		Присваивание значения элементу массива	
S151	Базовая поддержка объектов	Предикат типа	
S161	Расширенная поддержка объектов	Приведение подтипов	
S162		Приведение подтипов для ссылочных типов	
S202		Вызываемые из SQL подпрограммы, работающие с мультимножествами	
S231	Расширенная поддержка объектов	Указатели на структурные типы	
S232		Указатели на массивы	
S233		Указатели на мультимножества	
S241		Функции преобразований	
S242		Оператор изменения преобразования	
S251		Определяемые пользователем упорядочивания	
S261		Метод SPECIFICTYPE	
S271		Базовая поддержка мультимножеств	

Идентификатор	Пакет	Описание	Комментарий
S272		Мультимножества пользовательских типов	
S274		Мультимножества ссылочных типов	
S275		Расширенная поддержка мультимножеств	
S281		Типы вложенных коллекций	
S291		Ограничение уникальности для всей строки	
S401		Отдельные типы на базе типов массивов	
S402		Отдельные типы на базе отдельных типов	
S403		ARRAY_MAX_ CARDINALITY	
S404		TRIM_ARRAY	
T011		Тип TIMESTAMP в информационной схеме	
T021		Типы данных BINARY и VARBINARY	
T022		Расширенная поддержка типов данных BINARY и VARBINARY	
T023		Составные двоичные строки	
T024		Пробелы в двоичных строках	
T041	Базовая поддержка объектов	Базовая поддержка типа данных LOB	
T041-01	Базовая поддержка объектов	Тип данных BLOB	
T041-02	Базовая поддержка объектов	Тип данных CLOB	
T041-03	Базовая поддержка объектов	Функции POSITION, LENGTH, LOWER, TRIM, UPPER и SUBSTRING для типов данных LOB	
T041-04	Базовая поддержка объектов	Конкатенация типов данных LOB	
T041-05	Базовая поддержка объектов	Указатель на LOB: неудерживаемый	

Идентификатор	Пакет	Описание	Комментарий
T042		Расширенная поддержка типа данных LOB	
T043		Множитель T	
T044		Множитель P	
T051		Типы кортежей	
T052		MAX и MIN для типов кортежей	
T053		Явные псевдонимы ссылки на все поля	
T061		Поддержка UCS	
T101		Улучшенное определение возможности NULL	
T111		Изменяемые соединения, объединения и столбцы	
T175		Генерируемые столбцы	
T176		Поддержка генераторов последовательностей	
T180		Системное версионирование таблиц	
T181		Таблицы с периодом времени прикладного уровня	
T211	Активная база данных, улучшенное управление целостностью	Базовые возможности триггеров	
T211-06	Активная база данных, улучшенное управление целостностью	Поддержка правил времени выполнения для взаимодействия триггеров и ограничений	
T211-08	Активная база данных, улучшенное управление целостностью	Несколько триггеров для одного события вызываются в том порядке, в каком они были созданы в каталоге	намеренно опущено
T251		Оператор SET TRANSACTION: параметр LOCAL	
T261		Сцеплённые транзакции	

Идентификатор	Пакет	Описание	Комментарий
T272		Улучшенное управление точками сохранения	
T301		Функциональные зависимости	частично поддерживается
T321	Основа	Базовые вызываемые из SQL подпрограммы	
T321-02	Основа	Пользовательские хранимые процедуры без перегрузки	
T321-04	Основа	Оператор CALL	
T321-05	Основа	Оператор RETURN	
T322	PSM	Объявляемые атрибуты типа данных	
T324		Явное управление безопасностью подпрограмм SQL	
T326		Табличные функции	
T332		Расширенные роли	в основном поддерживаются
T434		GROUP BY DISTINCT	
T471		Наборы результатов в качестве возвращаемого значения	
T472		DESCRIBE CURSOR	
T495		Совместное изменение и извлечение данных	другой синтаксис
T502		Предикаты периодов	
T511		Счётчики транзакций	
T521		Именованные аргументы в операторе CALL	
T522		Значения по умолчанию для входных параметров процедур, вызываемых из SQL	поддерживаются, за исключением ключевого слова DEFAULT при вызове
T561		Удерживаемые указатели	
T571		Внешние вызываемые из SQL функции, возвращающие массивы	
T572		Внешние вызываемые из SQL функции, возвращающие мультимножества	

Идентификатор	Пакет	Описание	Комментарий
T601		Ссылки на локальные курсоры	
T612		Расширенные операции OLAP	поддерживаются некоторые формы
T616		Варианты обработки NULL для функций LEAD и LAG	
T618		Функция NTH_VALUE	функция существует, но некоторые возможности отсутствуют
T619		Вложенные оконные функции	
T620		Предложение WINDOW: параметр GROUPS	
T641		Присваивание нескольким столбцам	поддерживаются только некоторые варианты синтаксиса
T652		Операторы динамического SQL в SQL-подпрограммах	
T653		Операторы модификации схемы SQL во внешних подпрограммах	
T654		Операторы динамического SQL во внешних подпрограммах	
M001		Связи данных (DATALINK)	
M002		Связи данных через SQL/CLI	
M003		Связи данных через встроенный SQL	
M004		Поддержка сторонних данных	частично поддерживается
M005		Поддержка сторонних схем	
M006		Подпрограмма GetSQLString	
M007		TransmitRequest	
M009		Подпрограммы GetOpts и GetStatistics	
M010		Поддержка обёрток сторонних данных	другой API
M011		Связи данных через Ada	

Идентификатор	Пакет	Описание	Комментарий
M012		Связи данных через C	
M013		Связи данных через COBOL	
M014		Связи данных через Fortran	
M015		Связи данных через MUMPS	
M016		Связи данных через Pascal	
M017		Связи данных через PL/I	
M018		Подпрограммы интерфейса обёртки сторонних данных на языке Ada	
M019		Подпрограммы интерфейса обёртки сторонних данных на языке C	другой API
M020		Подпрограммы интерфейса обёртки сторонних данных на языке COBOL	
M021		Подпрограммы интерфейса обёртки сторонних данных на языке Fortran	
M022		Подпрограммы интерфейса обёртки сторонних данных на языке MUMPS	
M023		Подпрограммы интерфейса обёртки сторонних данных на языке Pascal	
M024		Подпрограммы интерфейса обёртки сторонних данных на языке PL/I	
M030		Поддержка сторонних данных SQL-сервера	
M031		Общие подпрограммы обёртки сторонних данных	
X012		Мультимножества типа XML	
X013		Отдельные типы, производные от XML	
X015		Поля типа XML	

Идентификатор	Пакет	Описание	Комментарий
X025		XMLCast	
X030		XMLDocument	
X038		XMLText	
X065		XMLParse: ввод BLOB и вариант CONTENT	
X066		XMLParse: ввод BLOB и вариант DOCUMENT	
X068		XMLSerialize: BOM	
X069		XMLSerialize: INDENT	
X073		XMLSerialize: сериализация BLOB и вариант CONTENT	
X074		XMLSerialize: сериализация BLOB и вариант DOCUMENT	
X075		XMLSerialize: сериализация BLOB	
X076		XMLSerialize: VERSION	
X077		XMLSerialize: явное указание ENCODING	
X078		XMLSerialize: явное объявление XML	
X080		Пространства имён при публикации XML	
X081		Объявления пространств имён XML на уровне запроса	
X082		Объявления пространств имён XML в DML	
X083		Объявления пространств имён XML в DDL	
X084		Объявления пространств имён XML в составных операторах	
X085		Предопределённые префиксы пространств имён	
X086		Объявления пространств имён XML в XMLTable	
X091		Предикат содержимого XML	
X096		XMLExists	только XPath 1.0
X100		Поддержка ведущего языка для XML: вариант CONTENT	

Идентификатор	Пакет	Описание	Комментарий
X101		Поддержка ведущего языка для XML: вариант DOCUMENT	
X110		Поддержка ведущего языка для XML: отображение VARCHAR	
X111		Поддержка ведущего языка для XML: отображение CLOB	
X112		Поддержка ведущего языка для XML: отображение BLOB	
X113		Поддержка ведущего языка для XML: указание STRIP WHITESPACE	
X114		Поддержка ведущего языка для XML: указание PRESERVE WHITESPACE	
X131		Предложение XMLBINARY на уровне запроса	
X132		Предложение XMLBINARY в DML	
X133		Предложение XMLBINARY в DDL	
X134		Предложение XMLBINARY в составных операторах	
X135		Предложение XMLBINARY в подзапросах	
X141		Предикат IS VALID: в зависимости от данных	
X142		Предикат IS VALID: предложение ACCORDING TO	
X143		Предикат IS VALID: предложение ELEMENT	
X144		Предикат IS VALID: расположение схемы	
X145		Предикат IS VALID вне ограничений-проверок	
X151		Предикат IS VALID с вариантом DOCUMENT	
X152		Предикат IS VALID с вариантом CONTENT	

Идентификатор	Пакет	Описание	Комментарий
X153		Предикат IS VALID с вариантом SEQUENCE	
X155		Предикат IS VALID: NAMESPACE без предложения ELEMENT	
X157		Предикат IS VALID: NO NAMESPACE с предложением ELEMENT	
X160		Базовая информационная схема для зарегистрированных XML-схем	
X161		Расширенная информационная схема для зарегистрированных XML-схем	
X170		Варианты обработки NULL с XML	
X171		Вариант NIL ON NO CONTENT	
X181		Тип XML(DOCUMENT(UNTYPED))	
X182		Тип XML(DOCUMENT(ANY))	
X190		Тип XML(SEQUENCE)	
X191		Тип XML(DOCUMENT(XMLSCHEMA))	
X192		Тип XML(CONTENT(XMLSCHEMA))	
X200		XMLQuery	
X201		XMLQuery: RETURNING CONTENT	
X202		XMLQuery: RETURNING SEQUENCE	
X203		XMLQuery: передача контекста	
X204		XMLQuery: инициализация переменной XQuery	
X205		XMLQuery: указание EMPTY ON EMPTY	
X206		XMLQuery: указание NULL ON EMPTY	
X211		Поддержка XML 1.1	

Идентификатор	Пакет	Описание	Комментарий
X221		Механизм передачи XML BY VALUE	
X231		Тип XML(CONTENT(UNTYPED))	
X232		Тип XML(CONTENT(ANY))	
X241		RETURNING CONTENT при публикации XML	
X242		RETURNING SEQUENCE при публикации XML	
X251		Хранимые значения XML типа XML(DOCUMENT(UNTYPED))	
X252		Хранимые значения XML типа XML(DOCUMENT(ANY))	
X253		Хранимые значения XML типа XML(CONTENT(UNTYPED))	
X254		Хранимые значения XML типа XML(CONTENT(ANY))	
X255		Хранимые значения XML типа XML(SEQUENCE)	
X256		Хранимые значения XML типа XML(DOCUMENT(XMLSCHEMA))	
X257		Хранимые значения XML типа XML(CONTENT(XMLSCHEMA))	
X260		Тип XML: предложение ELEMENT	
X261		Тип XML: NAMESPACE без предложения ELEMENT	
X263		Тип XML: NO NAMESPACE с предложением ELEMENT	
X264		Тип XML: расположение схемы	
X271		XMLValidate: в зависимости от данных	

Идентификатор	Пакет	Описание	Комментарий
X272		XMLValidate: предложение ACCORDING TO	
X273		XMLValidate: предложение ELEMENT	
X274		XMLValidate: расположение схемы	
X281		XMLValidate с вариантом DOCUMENT	
X282		XMLValidate с вариантом CONTENT	
X283		XMLValidate с вариантом SEQUENCE	
X284		XMLValidate: NAMESPACE без предложения ELEMENT	
X286		XMLValidate: NO NAMESPACE с предложением ELEMENT	
X300		XMLTable	только XPath 1.0
X305		XMLTable: инициализация переменной XQuery	

D.3. Ограничения XML и совместимость с SQL/XML

В SQL:2006 были внесены значительные изменения в посвящённой XML части ISO/IEC 9075-14 (SQL/XML). Реализация типа данных XML и связанных функций в Postgres Pro в большей степени соответствует более ранней редакции, SQL:2003, с некоторыми заимствованиями из последующих редакций. В частности:

- Тогда как в текущем стандарте существует семейство типов данных XML, содержащих «документы» или «содержимое» в нетипизированном виде или с типами XML Schema, а также тип `XML(SEQUENCE)`, содержащий произвольные части XML-документа, в Postgres Pro есть только один тип `xml`, который может содержать «документ» или «содержимое». Определённый в стандарте тип «последовательность» в Postgres Pro отсутствует.
- Postgres Pro предоставляет две функции, появившиеся в SQL:2006, но вместо языка XML Query, как должно быть согласно стандарту, в них используется язык XPath 1.0.

В этом разделе описаны некоторые из образовавшихся в итоге различий, с которыми вы можете столкнуться.

D.3.1. Запросы ограничиваются XPath версии 1.0

Специфичные для Postgres Pro функции `xpath()` и `xpath_exists()` выполняют запросы к XML-документам на языке XPath. В Postgres Pro также имеются поддерживающие только XPath стандартные функции `XMLEXISTS` и `XMLTABLE`, хотя согласно стандарту они должны поддерживать XQuery. Все эти функции в Postgres Pro реализованы с использованием библиотеки libxml2, которая поддерживает только XPath 1.0.

Существует тесная связь между языком XQuery и XPath версии 2.0 и новее: любое выражение, синтаксически правильное и выполняющееся успешно, выдаёт в обоих языках одинаковые результаты (за незначительным исключением, связанным с числовым обозначением символов или использованием предопределённых сущностей — XQuery заменяет их соответствующими символами, а XPath оставляет в исходном виде). Но между XPath 1.0 и этими языками подобная связь отсутствует: он появился гораздо раньше и во многом отличается от них.

Заслуживают отдельного рассмотрения две категории ограничений: ограничение языка XQuery до XPath для функций, описанных в стандарте SQL, и ограничение XPath до версии 1.0 как для стандартизированных функций, так и для специфичных функций Postgres Pro.

D.3.1.1. Ограничение языка XQuery до XPath

В число отличий XQuery от XPath входят:

- Выражения XQuery могут выдавать не только всевозможные значения XPath, но и конструировать новые XML-узлы. XPath может создавать и возвращать значения атомарных типов (числа, строки и так далее), но выдаваемые им XML-узлы должны уже присутствовать в документе, поступившем на вход выражения.
- В XQuery есть управляющие конструкции для организации циклов, сортировки и группировки.
- В XQuery поддерживается объявление и использование локальных функций.

В последних версиях XPath начинают появляться возможности, пересекающиеся с имеющимися в XQuery (например, конструкции `for-each`, `sort`, анонимные функции и функция `parse-xml`, создающая узел из строки), но до XPath 3.0 их не было.

D.3.1.2. Ограничения XPath до версии 1.0

Разработчикам, знакомым с XQuery и XPath 2.0 или новее, приходится иметь дело с рядом недостатков XPath версии 1.0:

- Фундаментальный тип результатов XQuery/XPath, тип `sequence`, который может содержать XML-узлы, атомарные значения, и всё это вместе, в XPath 1.0 отсутствует. В 1.0 выражения могут выдавать только набор узлов (состоящих из нуля или нескольких узлов XML) или единственное атомарное значение.
- В отличие от последовательностей XQuery/XPath, которые могут содержать произвольные элементы в любом требующемся порядке, во множестве узлов XPath 1.0 нет гарантированного порядка, и оно, как и любое другое множество, не может содержать несколько вхождений одного элемента.

> **Примечание**
>
> Библиотека libxml2 не всегда возвращает в Postgres Pro наборы узлов с внутренними членами в том порядке, в котором они идут во входном документе. В её документации не гарантируется корректное поведение, а выражение XPath 1.0 не может на это воздействовать.

- Тогда как XQuery/XPath поддерживают все типы, определённые в стандарте XML Schema, а также множество операторов и функций, работающих с этими типами, XPath 1.0 поддерживает только множества узлов и три атомарных типа: `boolean`, `double` и `string`.
- В XPath 1.0 отсутствует условный оператор. Выражение XQuery/XPath вида `if ( hat ) then hat/@size else "no hat"` не имеет эквивалента в XPath 1.0.
- В XPath 1.0 нет оператора сравнения строк с упорядочиванием. Условия `"cat" < "dog"` и `"cat" > "dog"` оба являются ложными, так как они выполняются как числовые сравнения двух значений `NaN`. Условия же `=` и `!=`, напротив, сравнивают строки в виде строк.
- XPath 1.0 размывает разницу между *сравнением значений* и *общими сравнениями*, которая имеется в XQuery/XPath. Сравнения `sale/@hatsize = 7` и `sale/@customer = "alice"` по сути

являются количественными сравнениями, и результатом их будет истина, если существует элемент `sale` с заданным значением атрибута, тогда как `sale/@taxable = false()` — сравнение всего набора узлов с *фактическим логическим значением*. Его результат будет истиной, только если у элемента `sale` вовсе не будет атрибута `taxable`.

- В модели данных XQuery/XPath *узел документа* может иметь либо форму документа (то есть содержать в точности один элемент верхнего уровня, снаружи которого допускаются только комментарии и инструкции обработки), либо форму содержимого (с ослабленными ограничениями). В XPath 1.0 ему соответствует *корневой узел*, который может иметь только форму документа. Этим отчасти объясняется то, что значение типа `xml`, передаваемое в качестве элемента контекста любым функциям Postgres Pro на базе XPath, должно быть в форме документа.

Кроме отмеченных выше имеются и другие различия. В языках XQuery и XPath версии 2.0 и новее существует режим совместимости с XPath 1.0, а в документации W3C имеется перечень *изменений функций* и *изменений в языке* применительно к этому режиму. Этот перечень гораздо более полный, но тоже не исчерпывающий. Даже режим совместимости этих языков не обеспечивает их полную идентичность XPath 1.0.

D.3.1.3. Преобразование значений/типов данных между SQL и XML

В SQL:2006 и более поздних ревизиях чётко определены преобразования между стандартными типами SQL и типами стандарта XML Schema в обе стороны. Однако эти правила выражаются в типах и понятиях, определённых в XQuery/XPath, и не могут быть непосредственно применены к другой модели данных, присущей XPath 1.0.

Когда Postgres Pro сопоставляет значения данных SQL с XML (как в функции `xmlelement`), или XML с SQL (как в выходных столбцах `xmltable`), за исключением нескольких отдельно обрабатываемых случаев, Postgres Pro просто полагает, что строка XPath 1.0, содержащая данные типа XML, будет допустимой для ввода в текстовом виде в тип данных SQL, и наоборот. Это правило добродетельно своей простотой, и при этом преобразования для многих типов данных в итоге оказываются такими, какими и должны быть согласно стандарту. В текущем выпуске требуется явное преобразование, если выражение столбца `xmltable` выдаёт логическое значение или число с плавающей точкой; см. Подраздел D.3.2.

Там же, где это нужно для взаимодействия с другими системами, для некоторых типов данных можно явно использовать функции форматирования типов данных (например, описанные в Разделе 9.8) для получения преобразований, в точности соответствующих стандарту.

D.3.2. Непреднамеренные ограничения реализации

В этом разделе описываются дополнительные ограничения, присущие текущей реализации в Postgres Pro, но не самой библиотеке libxml2.

D.3.2.1. Необходимость приведения столбцов `xmltable`, имеющих логический или числовой тип

Выражение столбца `xmltable`, результатом которого должно быть логическое или числовое значение XPath, будет выдавать ошибку «unexpected XPath object type» (неожиданный тип объекта XPath). Чтобы обойти эту ошибку, это выражение надо поместить внутрь вызова функции `string` языка XPath; в этом случае Postgres Pro сможет успешно присвоить строковое значение выходному столбцу SQL, имеющему логический или числовой тип с плавающей точкой.

D.3.2.2. Результат вычисления пути или выходной столбец SQL типа XML

В текущей версии выражение столбца `xmltable`, выдающее набор XML-узлов, может быть присвоено выходному SQL-столбцу типа XML. В этом случае результатом будет конкатенация следующих данных: для большинства типов узлов в наборе узлов берётся текстовый узел со *строковым значением* узла в определении XPath 1.0, но для узлов—элементов — копия этого узла. Такой набор узлов может быть присвоен SQL-столбцу, имеющему не XML-тип, только если этот

набор содержит один узел. При этом строковое значение узлов большинства типов заменяется пустой строкой, строковое значение узла-элемента заменяется конкатенацией только его непосредственных потомков — текстовых узлов (другие потомки исключаются из рассмотрения), а строковое значение текстового узла или атрибута выдаётся согласно определению XPath 1.0. Строковое выражение XPath, присваиваемое выходному столбцу типа XML, должно проходить разбор XML.

При разработке кода не рекомендуется полагаться на описанное особое поведение: во-первых, оно не вполне соответствует стандарту, а во-вторых, в Postgres Pro 12 оно изменено.

D.3.2.3. Передача параметров только по значению (`BY VALUE`)

В стандарте SQL определены два *механизма передачи параметров*, осуществляющих передачу XML-аргумента из SQL в XML-функцию или получение результата: `BY REF`, в котором конкретное значение в XML остаётся привязанным к своему узлу, и `BY VALUE`, в котором передаётся содержимое XML, но связь с узлом теряется. Выбрать механизм можно перед списком параметров, в качестве механизма по умолчанию для всех параметров, или после каждого отдельного параметра, переопределив тем самым выбор по умолчанию.

В качестве иллюстрации различия взгляните на следующие два запроса, которые в окружении SQL:2006 выдают true и false, если *x* является XML-значением:

```
SELECT XMLQUERY('$a is $b' PASSING BY REF x AS a, x AS b NULL ON EMPTY);
SELECT XMLQUERY('$a is $b' PASSING BY VALUE x AS a, x AS b NULL ON EMPTY);
```

В этом выпуске Postgres Pro принимает указание `BY REF` в конструкции `XMLEXISTS` или `XMLTABLE`, но игнорирует его. Тип `xml` содержит сериализованное представление данных в текстовом виде, поэтому сущность узла, которую нужно сохранять, отсутствует, и передача фактически производится по значению (`BY VALUE`).

D.3.2.4. Отсутствие именованных параметров запросов

Функции на базе XPath могут принимать один параметр, служащий контекстным элементом для выражения XPath, но не поддерживают передачу дополнительных значений, которые могли бы использоваться в выражении как именованные параметры.

D.3.2.5. Отсутствие типа `XML(SEQUENCE)`

Тип данных `xml` в Postgres Pro может содержать значение только в форме документа (`DOCUMENT`) или содержимого (`CONTENT`). Контекстный элемент выражения XQuery/XPath должен быть одиночным XML-узлом или атомарным значением, но в XPath 1.0 это может быть только XML-узел, и при этом нет типа узла, содержащего `CONTENT`. Как следствие, в Postgres Pro в качестве контекстного элемента XPath можно передать данные XML в единственном виде — в виде правильно оформленного документа (`DOCUMENT`).

Приложение E. Замечания к выпускам

В замечаниях к выпускам отмечаются значительные изменения, имевшие место в каждом выпуске Postgres Pro Standard, при этом ключевые изменения и вопросы миграции освещаются в самом начале. В эти замечания не включаются изменения, затрагивающие лишь нескольких пользователей, как и изменения внутренние и поэтому пользователям не заметные. Например, оптимизатор усовершенствуется почти в каждом выпуске, но для пользователей эти улучшения проявляются обычно просто в ускорении запросов.

E.1. Postgres Pro Standard 10.23.1

Дата выпуска: 2022-11-18

E.1.1. Обзор

Этот выпуск Postgres Pro Standard должен стать последним в серии 10. Пользователям следует поторопиться с переходом на более новую основную версию Postgres Pro Standard.

Этот выпуск основан на PostgreSQL 10.23 и Postgres Pro Standard 10.22.1. Все усовершенствования, унаследованные от PostgreSQL 10.23, перечислены в Замечаниях к выпуску PostgreSQL 10.23. Другие важные изменения и улучшения перечислены ниже:

- Приложение mamonsu обновлено до версии 3.5.2. Основные изменения перечислены ниже:
  - Добавлены метрики использования автоочистки.
  - Добавлены макросы Zabbix на основе параметров дополнительных модулей из файла конфигурации, так что теперь триггеры можно динамически менять прямо в Zabbix без обновления файла конфигурации и перезапуска mamonsu.
- Приложение pg_probackup обновлено до версии 2.5.10, в которой улучшена производительность и исправлены некоторые ошибки. Основные изменения перечислены ниже:
  - Устранена проблема в команде `checkdb --amcheck`, которая ранее не могла обрабатывать базы, содержащие секционированные индексы.
  - Устранена ошибка, возникавшая на ведомом сервере в случае, если на ведущем работал агент pg_probackup другой версии.

E.1.2. Миграция на версию 10.23.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Встроенные функции `numeric_eq`, `numeric_ne`, `numeric_gt`, `numeric_ge`, `numeric_lt` и `numeric_le` фактически являются герметичными, но они не имели соответствующей пометки в Postgres Pro Standard до версии 10.12.1, в результате чего запросы могли выполняться неоптимально. В частности, это отрицательно сказывалось на производительности запросов при использовании политик защиты на уровне строк. В версии 10.12.1 это было исправлено для новых инсталляций в результате корректировки исходных данных каталога, но существующие инсталляции сохранят некорректные пометки, пока вы не исправите записи этих функций в `pg_proc`. Чтобы их исправить, обновите экземпляр сервера с помощью pg_upgrade либо вручную измените их в существующем кластере, выполнив во всех его базах данных соответствующие команды `ALTER FUNCTION`. Например:

```
ALTER FUNCTION pg_catalog.numeric_eq LEAKPROOF
```

Начиная с Postgres Pro Standard 10.11.1, обновление версии ICU не препятствует запуску сервера. Перед подключением к базе данных, правило сортировки по умолчанию в которой использует ICU, Postgres Pro сравнивает версию этого правила сортировки с версией, предоставляемой библиотекой ICU, и выдаёт предупреждения в случае несовпадения версий; если вы полагаете, что изменение версии правила сортировки может повлиять на порядок сортировки ваших данных, перестройте объекты, использующие правило сортировки по умолчанию. Чтобы убрать эти предупреждения, вы можете выполнить команду `ALTER COLLATION "default" REFRESH VERSION`, как описано в ALTER COLLATION.

Так как в Postgres Pro Standard 10.7.1 была изменена модель распространения pg_probackup, при переходе с предыдущей версии в системе на базе Debian и Альт Линукс выполните `apt dist-upgrade` (или `apt-get dist-upgrade`), чтобы корректно обработались все новые зависимости. В Windows для завершения обновления вам нужно будет запустить отдельный инсталлятор pg_probackup.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям по миграции на версию 10. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.2. Postgres Pro Standard 10.22.1

Дата выпуска: 2022-08-22

E.2.1. Обзор

Этот выпуск основан на PostgreSQL 10.22 и Postgres Pro Standard 10.21.1. Все усовершенствования, унаследованные от PostgreSQL 10.22, перечислены в Замечаниях к выпуску PostgreSQL 10.22. Другие важные изменения и улучшения перечислены ниже:

- Приложение mamonsu обновлено до версии 3.5.1. Основные изменения перечислены ниже:
  - Устранена проблема безопасности, связанная с неограниченным доступом к файлу конфигурации, содержащему пароли. Теперь файл `agent.conf` доступен только пользователю `mamonsu`.
  - Исправлен дефект в расчёте метрик, который вызывал ошибки «float division by zero» (ошибка деления на ноль) в модуле STATEMENTS приложения mamonsu.
- Приложение pg_probackup обновлено до версии 2.5.7, в которой были исправлены некоторые ошибки и появились новые возможности:
  - Добавлен флаг `--dry-run` для команды `catchup`, позволяющий оценить размер передаваемых файлов данных без внесения изменений на диск.
  - Исправлена ошибка, препятствовавшая правильному перечитыванию блока после появления сообщения «File: ... blknum ... have wrong checksum, try again» (В файле: ... blknum ... содержит неверную контрольную сумму, попробуйте снова).
  - Реализована возможность вывода журналов в формате JSON, и добавлены соответствующие параметры для выбора формата
  - Добавлена возможность явного указания каталога, в который команда `restore` должна восстанавливать WAL.
  - Изменён уровень протоколирования некоторых сообщений pg_probackup для удобства чтения журналов пользователем.

- Ликвидировано зависание pg_probackup, которое могло произойти, если в файле конфигурации SSH `config` включался параметр `ControlMaster`. Теперь для параметра `ControlMaster` явно задаётся значение `no`.

E.2.2. Миграция на версию 10.22.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Встроенные функции `numeric_eq`, `numeric_ne`, `numeric_gt`, `numeric_ge`, `numeric_lt` и `numeric_le` фактически являются герметичными, но они не имели соответствующей пометки в Postgres Pro Standard до версии 10.12.1, в результате чего запросы могли выполняться неоптимально. В частности, это отрицательно сказывалось на производительности запросов при использовании политик защиты на уровне строк. В версии 10.12.1 это было исправлено для новых инсталляций в результате корректировки исходных данных каталога, но существующие инсталляции сохранят некорректные пометки, пока вы не исправите записи этих функций в `pg_proc`. Чтобы их исправить, обновите экземпляр сервера с помощью pg_upgrade либо вручную измените их в существующем кластере, выполнив во всех его базах данных соответствующие команды `ALTER FUNCTION`. Например:

```
ALTER FUNCTION pg_catalog.numeric_eq LEAKPROOF
```

Начиная с Postgres Pro Standard 10.11.1, обновление версии ICU не препятствует запуску сервера. Перед подключением к базе данных, правило сортировки по умолчанию в которой использует ICU, Postgres Pro сравнивает версию этого правила сортировки с версией, предоставляемой библиотекой ICU, и выдаёт предупреждения в случае несовпадения версий; если вы полагаете, что изменение версии правила сортировки может повлиять на порядок сортировки ваших данных, перестройте объекты, использующие правило сортировки по умолчанию. Чтобы убрать эти предупреждения, вы можете выполнить команду `ALTER COLLATION "default" REFRESH VERSION`, как описано в ALTER COLLATION.

Так как в Postgres Pro Standard 10.7.1 была изменена модель распространения pg_probackup, при переходе с предыдущей версии в системе на базе Debian и Альт Линукс выполните `apt dist-upgrade` (или `apt-get dist-upgrade`), чтобы корректно обработались все новые зависимости. В Windows для завершения обновления вам нужно будет запустить отдельный инсталлятор pg_probackup.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям по миграции на версию 10. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.3. Postgres Pro Standard 10.21.1

Дата выпуска: 2022-05-23

E.3.1. Обзор

Этот выпуск основан на PostgreSQL 10.21 и Postgres Pro Standard 10.20.1. Все усовершенствования, унаследованные от PostgreSQL 10.21, перечислены в Замечаниях к выпуску PostgreSQL 10.21. Другие важные изменения и улучшения перечислены ниже:

- Минимальные поддерживаемые версии Windows повышены до Windows 10 и Windows Server 2012 R2.
- Прекращена поддержка ОС Гослинукс 7.
- Прекращена поддержка системы Ubuntu 21.04.
- Модуль mamonsu обновлён до версии 3.4.0. Основные изменения перечислены ниже:
  - Добавлен новый параметр `timeout` в раздел `[zabbix]` конфигурационного файла mamonsu. Данный параметр позволяет установить максимальное время ожидания при подключении к серверу Zabbix.
  - Добавлена новая команда `mamonsu dashboard upload` *`имя_шаблона`*, предназначенная для загрузки панели Zabbix с метриками mamonsu в шаблон на сервере Zabbix версии 6.0 или выше.

E.3.2. Миграция на версию 10.21.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Встроенные функции `numeric_eq`, `numeric_ne`, `numeric_gt`, `numeric_ge`, `numeric_lt` и `numeric_le` фактически являются герметичными, но они не имели соответствующей пометки в Postgres Pro Standard до версии 10.12.1, в результате чего запросы могли выполняться неоптимально. В частности, это отрицательно сказывалось на производительности запросов при использовании политик защиты на уровне строк. В версии 10.12.1 это было исправлено для новых инсталляций в результате корректировки исходных данных каталога, но существующие инсталляции сохранят некорректные пометки, пока вы не исправите записи этих функций в `pg_proc`. Чтобы их исправить, обновите экземпляр сервера с помощью pg_upgrade либо вручную измените их в существующем кластере, выполнив во всех его базах данных соответствующие команды `ALTER FUNCTION`. Например:

```
ALTER FUNCTION pg_catalog.numeric_eq LEAKPROOF
```

Начиная с Postgres Pro Standard 10.11.1, обновление версии ICU не препятствует запуску сервера. Перед подключением к базе данных, правило сортировки по умолчанию в которой использует ICU, Postgres Pro сравнивает версию этого правила сортировки с версией, предоставляемой библиотекой ICU, и выдаёт предупреждения в случае несовпадения версий; если вы полагаете, что изменение версии правила сортировки может повлиять на порядок сортировки ваших данных, перестройте объекты, использующие правило сортировки по умолчанию. Чтобы убрать эти предупреждения, вы можете выполнить команду `ALTER COLLATION "default" REFRESH VERSION`, как описано в ALTER COLLATION.

Так как в Postgres Pro Standard 10.7.1 была изменена модель распространения pg_probackup, при переходе с предыдущей версии в системе на базе Debian и Альт Линукс выполните `apt dist-upgrade` (или `apt-get dist-upgrade`), чтобы корректно обработались все новые зависимости. В Windows для завершения обновления вам нужно будет запустить отдельный инсталлятор pg_probackup.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям по миграции на версию 10. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.4. Postgres Pro Standard 10.20.1

Дата выпуска: 2022-02-21

E.4.1. Обзор

Этот выпуск основан на PostgreSQL 10.20 и Postgres Pro Standard 10.19.1. Все усовершенствования, унаследованные от PostgreSQL 10.20, перечислены в Замечаниях к выпуску PostgreSQL 10.20. Другие важные изменения и улучшения перечислены ниже:

- Исправлены параметры уничтожения процессов при нехватке памяти (OOM) в конфигурации службы Postgres Pro. Ранее поправка очков OOM устанавливалась для всех процессов Postgres Pro, а не только для postmaster, поэтому, когда Postgres Pro исчерпывал всю оперативную память, могли уничтожаться другие важные процессы, например, sshd.
- Прекращена поддержка устаревших операционных систем Debian 8, Astra Linux «Смоленск» 1.5 и МСВСфера 6.3.
- Приложение pg_probackup обновлено до версии 2.5.5, в которой по сравнению с версией 2.5.3 были исправлены некоторые ошибки и появились новые возможности:
  - Для команды `checkdb` добавлен параметр `--checkunique` (работающий только с `--amcheck`), позволяющий проверять ограничения уникальности в ходе проверки индексов. Для соответствующей проверки в базе данных должно быть установлено расширение amcheck подходящей версии. За подробностями обратитесь к документации amcheck.
  - Устранена проблема, которая могла возникнуть, когда исходная БД не находилась на линии времени 1 и целевая не отставала от исходной: probackup проверял целостность истории линии времени `catchup` и выводил сообщение об ошибке «Destination is not in source timeline history» (Целевого состояния нет в истории исходной линии времени).
  - Устранена проблема, связанная с поведением `catchup` в режимах `DELTA` и `PTRACK`. Она могла проявляться, когда использовалось удалённое подключение через SSH и в параметрах `--destination-pgdata` и `--source-pgdata` задавался одинаковый путь, и вызывала повреждение исходного экземпляра (как минимум удалялись файлы `global/pg_control`, `global/pg_filenode.map` и `base/*/pg_filenode.map`); в результате выводилось сообщение вида «Could not open file "/pgwal/test/global/pg_control" for reading: No such file or directory» («Не удалось открыть для чтения файл "/pgwal/test/global/pg_control": нет такого файла или каталога»).

E.4.2. Миграция на версию 10.20.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Встроенные функции `numeric_eq`, `numeric_ne`, `numeric_gt`, `numeric_ge`, `numeric_lt` и `numeric_le` фактически являются герметичными, но они не имели соответствующей пометки в Postgres Pro Standard до версии 10.12.1, в результате чего запросы могли выполняться неоптимально. В частности, это отрицательно сказывалось на производительности запросов при использовании политик защиты на уровне строк. В версии 10.12.1 это было исправлено для новых инсталляций в результате корректировки исходных данных каталога, но существующие инсталляции сохранят некорректные пометки, пока вы не исправите записи этих функций в `pg_proc`. Чтобы их исправить, обновите экземпляр сервера с помощью pg_upgrade либо вручную измените их в существующем кластере, выполнив во всех его базах данных соответствующие команды `ALTER FUNCTION`. Например:

```
ALTER FUNCTION pg_catalog.numeric_eq LEAKPROOF
```

Начиная с Postgres Pro Standard 10.11.1, обновление версии ICU не препятствует запуску сервера. Перед подключением к базе данных, правило сортировки по умолчанию в которой использует

ICU, Postgres Pro сравнивает версию этого правила сортировки с версией, предоставляемой библиотекой ICU, и выдаёт предупреждения в случае несовпадения версий; если вы полагаете, что изменение версии правила сортировки может повлиять на порядок сортировки ваших данных, перестройте объекты, использующие правило сортировки по умолчанию. Чтобы убрать эти предупреждения, вы можете выполнить команду `ALTER COLLATION "default" REFRESH VERSION`, как описано в ALTER COLLATION.

Так как в Postgres Pro Standard 10.7.1 была изменена модель распространения pg_probackup, при переходе с предыдущей версии в системе на базе Debian и Альт Линукс выполните `apt dist-upgrade` (или `apt-get dist-upgrade`), чтобы корректно обработались все новые зависимости. В Windows для завершения обновления вам нужно будет запустить отдельный инсталлятор pg_probackup.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям по миграции на версию 10. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.5. Postgres Pro Standard 10.19.1

Дата выпуска: 2021-11-23

E.5.1. Обзор

Этот выпуск основан на PostgreSQL 10.19 и Postgres Pro Standard 10.18.1. Все усовершенствования, унаследованные от PostgreSQL 10.19, перечислены в Замечаниях к выпуску PostgreSQL 10.19. Другие важные изменения и улучшения перечислены ниже:

- Прекращена поддержка расширения sr_plan.
- Модуль mamonsu обновлён до версии 3.1.0. Основные изменения перечислены ниже:
  - У команды `bootstrap` появилась новая возможность. Благодаря ей, если параметр `-dbname` не указан явно, он автоматически примет значение, соответствующее имени базы данных, которое указано в разделе `[postgres]` файла конфигурации mamonsu.
  - Исправлена критическая проблема, которая могла возникнуть при пропуске шага `bootstrap`. В этом случае, хотя пользователь мог работать с mamonsu с правами суперпользователя, специальная схема `mamonsu` не создавалась, но после создания вспомогательного расширения pg_buffercache в любой другой схеме запуск mamonsu приводил к ошибкам «`ERROR: relation "mamonsu.pg_buffercache" does not exist`» (ОШИБКА: отношение "mamonsu.pg_buffercache" не существует).
- Добавлена поддержка ОС Альт Линукс 10.
- Прекращена поддержка системы Ubuntu 20.10.
- Решена проблема pg_probackup с архивированием WAL из каталога за пределами каталога данных. Ранее многопоточная и пакетная оптимизации команды `archive-push` препятствовали такому архивированию; теперь можно архивировать WAL из любого каталога, но если он находится за пределами каталога данных, многопоточная и пакетная оптимизации автоматически отключаются.
- В pg_dumpall устранена возможность использования небезопасного пути поиска.
- Оператор `WAITLSN` признан устаревшим.

E.5.2. Миграция на версию 10.19.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Встроенные функции `numeric_eq`, `numeric_ne`, `numeric_gt`, `numeric_ge`, `numeric_lt` и `numeric_le` фактически являются герметичными, но они не имели соответствующей пометки в Postgres Pro Standard до версии 10.12.1, в результате чего запросы могли выполняться неоптимально. В частности, это отрицательно сказывалось на производительности запросов при использовании политик защиты на уровне строк. В версии 10.12.1 это было исправлено для новых инсталляций в результате корректировки исходных данных каталога, но существующие инсталляции сохранят некорректные пометки, пока вы не исправите записи этих функций в `pg_proc`. Чтобы их исправить, обновите экземпляр сервера с помощью pg_upgrade либо вручную измените их в существующем кластере, выполнив во всех его базах данных соответствующие команды `ALTER FUNCTION`. Например:

```
ALTER FUNCTION pg_catalog.numeric_eq LEAKPROOF
```

Начиная с Postgres Pro Standard 10.11.1, обновление версии ICU не препятствует запуску сервера. Перед подключением к базе данных, правило сортировки по умолчанию в которой использует ICU, Postgres Pro сравнивает версию этого правила сортировки с версией, предоставляемой библиотекой ICU, и выдаёт предупреждения в случае несовпадения версий; если вы полагаете, что изменение версии правила сортировки может повлиять на порядок сортировки ваших данных, перестройте объекты, использующие правило сортировки по умолчанию. Чтобы убрать эти предупреждения, вы можете выполнить команду `ALTER COLLATION "default" REFRESH VERSION`, как описано в ALTER COLLATION.

Так как в Postgres Pro Standard 10.7.1 была изменена модель распространения pg_probackup, при переходе с предыдущей версии в системе на базе Debian и Альт Линукс выполните `apt dist-upgrade` (или `apt-get dist-upgrade`), чтобы корректно обработались все новые зависимости. В Windows для завершения обновления вам нужно будет запустить отдельный инсталлятор pg_probackup.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям по миграции на версию 10. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.6. Postgres Pro Standard 10.18.1

Дата выпуска: 2021-08-23

E.6.1. Обзор

Этот выпуск основан на PostgreSQL 10.18 и Postgres Pro Standard 10.17.1. Все усовершенствования, унаследованные от PostgreSQL 10.18, перечислены в Замечаниях к выпуску PostgreSQL 10.18. Другие важные изменения и улучшения перечислены ниже:

- Приложение mamonsu обновлено до версии 3.0, которая несовместима с предыдущей. Прежде чем перейти к использованию данной версии, ознакомьтесь с Замечаниями о совместимости mamonsu.
- Добавлена поддержка операционной системы Astra Linux «Смоленск» 1.7.

- Исправлено поведение команды `ALTER TABLE IF EXISTS` *`таблица`* `RENAME TO` *`таблица1`* в случаях, когда исходная таблица отсутствует и при этом загружено расширение pg_pathman. Ранее выдавалось сообщение уровня `ERROR`, а теперь — `NOTICE`; именно так ведёт себя Postgres Pro, когда расширение pg_pathman не загружено.
- Приложение pg_probackup обновлено до версии 2.5.1, в которой добавлена новая команда `catchup` для копирования экземпляра Postgres Pro напрямую, без использования каталога резервных копий. Это позволяет добавить новый ведомый сервер в параллельном режиме или синхронизировать отставший ведомый сервер с ведущим.
- Исправлена ошибка в оптимизаторе запросов, в результате которой завышалась стоимость сканирования уникального индекса. Это проявлялось в том, что в случае выбора при сканировании между уникальным индексом и неуникальным предпочитался второй, что могло приводить к чрезмерному количеству обращений к диску и общему замедлению.

E.6.2. Миграция на версию 10.18.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Встроенные функции `numeric_eq`, `numeric_ne`, `numeric_gt`, `numeric_ge`, `numeric_lt` и `numeric_le` фактически являются герметичными, но они не имели соответствующей пометки в Postgres Pro Standard до версии 10.12.1, в результате чего запросы могли выполняться неоптимально. В частности, это отрицательно сказывалось на производительности запросов при использовании политик защиты на уровне строк. В версии 10.12.1 это было исправлено для новых инсталляций в результате корректировки исходных данных каталога, но существующие инсталляции сохранят некорректные пометки, пока вы не исправите записи этих функций в `pg_proc`. Чтобы их исправить, обновите экземпляр сервера с помощью pg_upgrade либо вручную измените их в существующем кластере, выполнив во всех его базах данных соответствующие команды `ALTER FUNCTION`. Например:

```
ALTER FUNCTION pg_catalog.numeric_eq LEAKPROOF
```

Начиная с Postgres Pro Standard 10.11.1, обновление версии ICU не препятствует запуску сервера. Перед подключением к базе данных, правило сортировки по умолчанию в которой использует ICU, Postgres Pro сравнивает версию этого правила сортировки с версией, предоставляемой библиотекой ICU, и выдаёт предупреждения в случае несовпадения версий; если вы полагаете, что изменение версии правила сортировки может повлиять на порядок сортировки ваших данных, перестройте объекты, использующие правило сортировки по умолчанию. Чтобы убрать эти предупреждения, вы можете выполнить команду `ALTER COLLATION "default" REFRESH VERSION`, как описано в ALTER COLLATION.

Так как в Postgres Pro Standard 10.7.1 была изменена модель распространения pg_probackup, при переходе с предыдущей версии в системе на базе Debian и Альт Линукс выполните `apt dist-upgrade` (или `apt-get dist-upgrade`), чтобы корректно обработались все новые зависимости. В Windows для завершения обновления вам нужно будет запустить отдельный инсталлятор pg_probackup.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям по миграции на версию 10. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.7. Postgres Pro Standard 10.17.1

Дата выпуска: 2021-05-25

E.7.1. Обзор

Этот выпуск основан на PostgreSQL 10.17 и Postgres Pro Standard 10.6.1. Все усовершенствования, унаследованные от PostgreSQL 10.17, перечислены в Замечаниях к выпуску PostgreSQL 10.17. Другие важные изменения и улучшения перечислены ниже:

- Удалён словарь Hunspell `hunspell_ne_np`. При необходимости используйте расширение, опубликованное в *github*.
- Добавлена поддержка ОС Ubuntu 21.04; поддержка Ubuntu 16.04 прекращена.
- Решена проблема установки Postgres Pro в Альт 8.2 СП. Теперь для этой ОС предоставляется отдельный репозиторий пакетов, отличный от репозиториев для младших версий Альт СП.
- Приложение mamonsu обновлено до версии 2.7.0.
- Утилита pgpro_controldata обновлена до версии 13.2.0.

E.7.2. Миграция на версию 10.17.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Встроенные функции `numeric_eq`, `numeric_ne`, `numeric_gt`, `numeric_ge`, `numeric_lt` и `numeric_le` фактически являются герметичными, но они не имели соответствующей пометки в Postgres Pro Standard до версии 10.12.1, в результате чего запросы могли выполняться неоптимально. В частности, это отрицательно сказывалось на производительности запросов при использовании политик защиты на уровне строк. В версии 10.12.1 это было исправлено для новых инсталляций в результате корректировки исходных данных каталога, но существующие инсталляции сохранят некорректные пометки, пока вы не исправите записи этих функций в `pg_proc`. Чтобы их исправить, обновите экземпляр сервера с помощью pg_upgrade либо вручную измените их в существующем кластере, выполнив во всех его базах данных соответствующие команды `ALTER FUNCTION`. Например:

```
ALTER FUNCTION pg_catalog.numeric_eq LEAKPROOF
```

Начиная с Postgres Pro Standard 10.11.1, обновление версии ICU не препятствует запуску сервера. Перед подключением к базе данных, правило сортировки по умолчанию в которой использует ICU, Postgres Pro сравнивает версию этого правила сортировки с версией, предоставляемой библиотекой ICU, и выдаёт предупреждения в случае несовпадения версий; если вы полагаете, что изменение версии правила сортировки может повлиять на порядок сортировки ваших данных, перестройте объекты, использующие правило сортировки по умолчанию. Чтобы убрать эти предупреждения, вы можете выполнить команду `ALTER COLLATION "default" REFRESH VERSION`, как описано в ALTER COLLATION.

Так как в Postgres Pro Standard 10.7.1 была изменена модель распространения pg_probackup, при переходе с предыдущей версии в системе на базе Debian и Альт Линукс выполните `apt dist-upgrade` (или `apt-get dist-upgrade`), чтобы корректно обработались все новые зависимости. В Windows для завершения обновления вам нужно будет запустить отдельный инсталлятор pg_probackup.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям по миграции на версию 10. Если вы

выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.8. Postgres Pro Standard 10.16.1

Дата выпуска: 2021-02-19

E.8.1. Обзор

Этот выпуск основан на PostgreSQL 10.16 и Postgres Pro Standard 10.15.1. Все усовершенствования, унаследованные от PostgreSQL 10.16, перечислены в Замечаниях к выпуску PostgreSQL 10.16. Другие важные изменения и улучшения перечислены ниже:

- Приложение pg_probackup обновлено до версии 2.4.10. Её основные отличия от ранее поставляемой версии 2.4.2:
  - При инкрементальном восстановлении с флагом `--force` можно перезаписать содержимое каталога, задаваемого параметром `PGDATA`, в случае совпадения системных идентификаторов. Ранее в подобных случаях выдавалась ошибка.
  - Стало возможным восстанавливать или проверять копии, находящиеся в файловой системе «только для чтения».
  - Объединение копий на месте теперь не поддерживается только при смене формата хранения.
  - Реализованы разделяемые блокировки копий, что позволяет выполнять проверку и восстановление сразу нескольких копий. Эти блокировки освобождаются при завершении программы.
  - Передаваемые сегменты WAL теперь добавляются в список файлов копии «на лету» и сбрасываются на диск в конце резервного копирования.

  За подробностями обратитесь к документации pg_probackup.
- Добавлена утилита pgpro_controldata, показывающая управляющую информацию кластера БД PostgreSQL/Postgres Pro и параметры совместимости кластера и/или сервера.
- Добавлена функция `pg_snapshot_any`, позволяющая суперпользователям исследовать повреждённые базы данных. За подробностями обратитесь к Подразделу 9.26.11.

E.8.2. Миграция на версию 10.16.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Встроенные функции `numeric_eq`, `numeric_ne`, `numeric_gt`, `numeric_ge`, `numeric_lt` и `numeric_le` фактически являются герметичными, но они не имели соответствующей пометки в Postgres Pro Standard до версии 10.12.1, в результате чего запросы могли выполняться неоптимально. В частности, это отрицательно сказывалось на производительности запросов при использовании политик защиты на уровне строк. В версии 10.12.1 это было исправлено для новых инсталляций в результате корректировки исходных данных каталога, но существующие инсталляции сохранят некорректные пометки, пока вы не исправите записи этих функций в `pg_proc`. Чтобы их исправить, обновите экземпляр сервера с помощью pg_upgrade либо вручную измените их в существующем кластере, выполнив во всех его базах данных соответствующие команды `ALTER FUNCTION`. Например:

```
ALTER FUNCTION pg_catalog.numeric_eq LEAKPROOF
```

Начиная с Postgres Pro Standard 10.11.1, обновление версии ICU не препятствует запуску сервера. Перед подключением к базе данных, правило сортировки по умолчанию в которой использует

ICU, Postgres Pro сравнивает версию этого правила сортировки с версией, предоставляемой библиотекой ICU, и выдаёт предупреждения в случае несовпадения версий; если вы полагаете, что изменение версии правила сортировки может повлиять на порядок сортировки ваших данных, перестройте объекты, использующие правило сортировки по умолчанию. Чтобы убрать эти предупреждения, вы можете выполнить команду `ALTER COLLATION "default" REFRESH VERSION`, как описано в ALTER COLLATION.

Так как в Postgres Pro Standard 10.7.1 была изменена модель распространения pg_probackup, при переходе с предыдущей версии в системе на базе Debian и Альт Линукс выполните `apt dist-upgrade` (или `apt-get dist-upgrade`), чтобы корректно обработались все новые зависимости. В Windows для завершения обновления вам нужно будет запустить отдельный инсталлятор pg_probackup.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям по миграции на версию 10. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.9. Postgres Pro Standard 10.15.1

Дата выпуска: 2020-11-23

E.9.1. Обзор

Этот выпуск основан на PostgreSQL 10.15 и Postgres Pro Standard 10.14.1. Все усовершенствования, унаследованные от PostgreSQL 10.15, перечислены в Замечаниях к выпуску PostgreSQL 10.15. Другие важные изменения и улучшения перечислены ниже:

- Добавлена поддержка ОС Ubuntu 20.10; поддержка Ubuntu 19.10 прекращена.
- Прекращена поддержка ОС Red Hat Enterprise Linux (RHEL) 6 и основанных на ней Oracle Linux 6 и CentOS 6.
- Исправлена ошибка в модуле pg_variables. Теперь вы можете создавать в транзакции переменные после удаления одноимённых.
- Приложение mamonsu обновлено до версии 2.6.1.
- pgbouncer обновлён до версии 1.15.

E.9.2. Миграция на версию 10.15.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Встроенные функции `numeric_eq`, `numeric_ne`, `numeric_gt`, `numeric_ge`, `numeric_lt` и `numeric_le` фактически являются герметичными, но они не имели соответствующей пометки в Postgres Pro Standard до версии 10.12.1, в результате чего запросы могли выполняться неоптимально. В частности, это отрицательно сказывалось на производительности запросов при использовании политик защиты на уровне строк. В версии 10.12.1 это было исправлено для новых инсталляций в результате корректировки исходных данных каталога, но существующие инсталляции сохранят некорректные пометки, пока вы не исправите записи этих функций в `pg_proc`. Чтобы их исправить, обновите экземпляр сервера с помощью pg_upgrade либо вручную измените их в существующем

кластере, выполнив во всех его базах данных соответствующие команды `ALTER FUNCTION`. Например:

```
ALTER FUNCTION pg_catalog.numeric_eq LEAKPROOF
```

Начиная с Postgres Pro Standard 10.11.1, обновление версии ICU не препятствует запуску сервера. Перед подключением к базе данных, правило сортировки по умолчанию в которой использует ICU, Postgres Pro сравнивает версию этого правила сортировки с версией, предоставляемой библиотекой ICU, и выдаёт предупреждения в случае несовпадения версий; если вы полагаете, что изменение версии правила сортировки может повлиять на порядок сортировки ваших данных, перестройте объекты, использующие правило сортировки по умолчанию. Чтобы убрать эти предупреждения, вы можете выполнить команду `ALTER COLLATION "default" REFRESH VERSION`, как описано в ALTER COLLATION.

Так как в Postgres Pro Standard 10.7.1 была изменена модель распространения pg_probackup, при переходе с предыдущей версии в системе на базе Debian и Альт Линукс выполните `apt dist-upgrade` (или `apt-get dist-upgrade`), чтобы корректно обработались все новые зависимости. В Windows для завершения обновления вам нужно будет запустить отдельный инсталлятор pg_probackup.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям по миграции на версию 10. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.10. Postgres Pro Standard 10.14.1

Дата выпуска: 2020-08-25

E.10.1. Обзор

Этот выпуск основан на PostgreSQL 10.14 и Postgres Pro Standard 10.13.1. Все усовершенствования, унаследованные от PostgreSQL 10.14, перечислены в Замечаниях к выпуску PostgreSQL 10.14. Другие важные изменения и улучшения перечислены ниже:

- Приложение pg_probackup обновлено до версии 2.4.2:
  - Для большей гибкости добавлены новые параметры и флаги команд `delete`, `backup`, `restore`, `archive-push` и `set-backup`.
  - Добавлена возможность инкрементального восстановления и поддержка инкрементальных цепочек с разными линиями времени.
  - Во время восстановления теперь могут задаваться параметры Postgres Pro *`slot_name`* и *`primary_conninfo`*.
  - Существенно переработаны команды `archive-push` и `archive-get`.
  - Были улучшены показатели производительности и потребления памяти.

  За подробностями обратитесь к документации pg_probackup.
- Добавлен необязательный аргумент `tune` команды `pg-setup initdb`. С помощью этого аргумента вы можете выбрать для вашего кластера одну из предопределённых настроенных конфигураций.
- Прекращена поддержка Альт Линукс 7, при этом поддержка Альт Линукс СПТ 7.0 сохранена ради удобства существующих клиентов.

- Прекращена поддержка SUSE Linux Enterprise Server 11.
- Исправлены дефекты в реализации BRIN, приводившие к следующим ошибкам:
  - «failed to find parent tuple for heap-only tuple ...» (не удалось найти родительский кортеж для неиндексированного кортежа...)

    Эта ошибка могла возникать, когда в параллельных транзакциях одновременно выполнялись функция `brin_summarize_new_values()` и изменения данных по методу HOT.
  - «corrupted BRIN index: inconsistent range map» (испорченный индекс BRIN: несогласованность в карте диапазонов)

    Эта ошибка могла возникать, когда в параллельных транзакциях одновременно выполнялись сброс обобщения зоны страниц и сканирование индекса.
- Приложение mamonsu для Linux обновлено до версии 2.5.1. Теперь оно базируется на Python 3. Для Windows по-прежнему поставляется версия 2.3.4.

E.10.2. Миграция на версию 10.14.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Встроенные функции `numeric_eq`, `numeric_ne`, `numeric_gt`, `numeric_ge`, `numeric_lt` и `numeric_le` фактически являются герметичными, но они не имели соответствующей пометки в Postgres Pro Standard до версии 10.12.1, в результате чего запросы могли выполняться неоптимально. В частности, это отрицательно сказывалось на производительности запросов при использовании политик защиты на уровне строк. В версии 10.12.1 это было исправлено для новых инсталляций в результате корректировки исходных данных каталога, но существующие инсталляции сохранят некорректные пометки, пока вы не исправите записи этих функций в `pg_proc`. Чтобы их исправить, обновите экземпляр сервера с помощью pg_upgrade либо вручную измените их в существующем кластере, выполнив во всех его базах данных соответствующие команды `ALTER FUNCTION`. Например:

```
ALTER FUNCTION pg_catalog.numeric_eq LEAKPROOF
```

Начиная с Postgres Pro Standard 10.11.1, обновление версии ICU не препятствует запуску сервера. Перед подключением к базе данных, правило сортировки по умолчанию в которой использует ICU, Postgres Pro сравнивает версию этого правила сортировки с версией, предоставляемой библиотекой ICU, и выдаёт предупреждения в случае несовпадения версий; если вы полагаете, что изменение версии правила сортировки может повлиять на порядок сортировки ваших данных, перестройте объекты, использующие правило сортировки по умолчанию. Чтобы убрать эти предупреждения, вы можете выполнить команду `ALTER COLLATION "default" REFRESH VERSION`, как описано в ALTER COLLATION.

Так как в Postgres Pro Standard 10.7.1 была изменена модель распространения pg_probackup, при переходе с предыдущей версии в системе на базе Debian и Альт Линукс выполните `apt dist-upgrade` (или `apt-get dist-upgrade`), чтобы корректно обработались все новые зависимости. В Windows для завершения обновления вам нужно будет запустить отдельный инсталлятор pg_probackup.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям по миграции на версию 10. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр

`--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.11. Postgres Pro Standard 10.13.1

Дата выпуска: 2020-05-21

E.11.1. Обзор

Этот выпуск основан на PostgreSQL 10.13 и Postgres Pro Standard 10.12.1. Все усовершенствования, унаследованные от PostgreSQL 10.13, перечислены в Замечаниях к выпуску PostgreSQL 10.13. Другие важные изменения и улучшения перечислены ниже:

- Устранена утечка памяти, проявлявшаяся при операциях соединения, в которых требовалось оценивать множество путей выполнения для сложных запросов.
- Добавлен параметр `--no-data-checksums` для initdb, позволяющий инициализировать кластеры Postgres Pro с отключёнными контрольными суммами.
- Добавлена поддержка Oracle Linux 8.
- Добавлена поддержка ОС Альт 8 СП.
- Добавлена поддержка ОС Ubuntu 20.04; поддержка Ubuntu 19.04 прекращена.
- Добавлен пакет `plpython3` для ОС Rosa Enterprise Linux Server 7 и РОСА «КОБАЛЬТ» (серверная редакция) на платформе РОСА 7, а также для РЕД ОС 7 МУРОМ, Гослинукс 7 и AlterOS 7.5, где его раньше не было. Теперь он предоставляется для всех поддерживаемых платформ, кроме SLES 11 и МСВСфера 6.3.
- Обновлено несколько библиотек, поставляемых с Postgres Pro для Windows, до следующих версий: OpenSSL 1.1.1g, libzstd 1.4.4, gettext 0.20.2, libiconv 1.16, libxml2 2.9.9 и libxslt 1.1.32. Библиотека ICU была обновлена с версии 56.1 до 56.2.
- Расширение pg_pathman обновлено до версии 1.5.11. В данной версии устранена ошибка, возникавшая при попытке обращения к дочерним секциям, к которым нет доступа, при том, что к родительской таблице доступ разрешён. Эта проблема могла наблюдаться ранее при использовании pg_pathman с Postgres Pro 11.7.1 или новее.

E.11.2. Миграция на версию 10.13.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Встроенные функции `numeric_eq`, `numeric_ne`, `numeric_gt`, `numeric_ge`, `numeric_lt` и `numeric_le` фактически являются герметичными, но они не имели соответствующей пометки в Postgres Pro Standard до версии 10.12.1, в результате чего запросы могли выполняться неоптимально. В частности, это отрицательно сказывалось на производительности запросов при использовании политик защиты на уровне строк. В версии 10.12.1 это было исправлено для новых инсталляций в результате корректировки исходных данных каталога, но существующие инсталляции сохранят некорректные пометки, пока вы не исправите записи этих функций в `pg_proc`. Чтобы их исправить, обновите экземпляр сервера с помощью pg_upgrade либо вручную измените их в существующем кластере, выполнив во всех его базах данных соответствующие команды `ALTER FUNCTION`. Например:

```
ALTER FUNCTION pg_catalog.numeric_eq LEAKPROOF
```

Начиная с Postgres Pro Standard 10.11.1, обновление версии ICU не препятствует запуску сервера. Перед подключением к базе данных, правило сортировки по умолчанию в которой использует ICU, Postgres Pro сравнивает версию этого правила сортировки с версией, предоставляемой библиотекой ICU, и выдаёт предупреждения в случае несовпадения версий; если вы полагаете, что изменение версии правила сортировки может повлиять на порядок сортировки ваших данных,

перестройте объекты, использующие правило сортировки по умолчанию. Чтобы убрать эти предупреждения, вы можете выполнить команду `ALTER COLLATION "default" REFRESH VERSION`, как описано в ALTER COLLATION.

Так как в Postgres Pro Standard 10.7.1 была изменена модель распространения pg_probackup, при переходе с предыдущей версии в системе на базе Debian и Альт Линукс выполните `apt dist-upgrade` (или `apt-get dist-upgrade`), чтобы корректно обработались все новые зависимости. В Windows для завершения обновления вам нужно будет запустить отдельный инсталлятор pg_probackup.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям по миграции на версию 10. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.12. Postgres Pro Standard 10.12.1

Дата выпуска: 2020-02-26

E.12.1. Обзор

Этот выпуск основан на PostgreSQL 10.12 и Postgres Pro Standard 10.11.1. Все усовершенствования, унаследованные от PostgreSQL 10.12, перечислены в Замечаниях к выпуску PostgreSQL 10.12. Другие важные изменения и улучшения перечислены ниже:

- Максимально допустимое значение для параметра track_activity_query_size увеличено до 1 МБ. В ванильном PostgreSQL это изменение запланировано на версию 13.
- Исправлена оптимизация планировщика, призванная исключать идентичные выражения `OR`, чтобы не исключались похожие выражения, в которых фигурируют разные столбцы, но занимающие одинаковые позиции в разных индексах.
- В модуле `mchar` исправлена обработка предложения `ESCAPE` в регулярных выражениях `SIMILAR TO`.
- Улучшена производительность запросов с включённой политикой защиты на уровне строк в результате признания функций численных сравнений герметичными.
- Приложение pg_probackup обновлено до версии 2.2.7.
- Приложение mamonsu обновлено до версии 2.4.4.

E.12.2. Миграция на версию 10.12.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Встроенные функции `numeric_eq`, `numeric_ne`, `numeric_gt`, `numeric_ge`, `numeric_lt` и `numeric_le` фактически являются герметичными, но они не имели соответствующей пометки в Postgres Pro Standard до версии 10.12.1, в результате чего запросы могли выполняться неоптимально. В частности, это отрицательно сказывалось на производительности запросов при использовании политик защиты на уровне строк. В версии 10.12.1 это было исправлено для новых инсталляций в результате корректировки исходных данных каталога, но существующие инсталляции сохранят некорректные пометки, пока вы не исправите записи этих функций в `pg_proc`. Чтобы их исправить,

обновите экземпляр сервера с помощью pg_upgrade либо вручную измените их в существующем кластере, выполнив во всех его базах данных соответствующие команды `ALTER FUNCTION`. Например:

```
ALTER FUNCTION pg_catalog.numeric_eq LEAKPROOF
```

Начиная с Postgres Pro Standard 10.11.1, обновление версии ICU не препятствует запуску сервера. Перед подключением к базе данных, правило сортировки по умолчанию в которой использует ICU, Postgres Pro сравнивает версию этого правила сортировки с версией, предоставляемой библиотекой ICU, и выдаёт предупреждения в случае несовпадения версий; если вы полагаете, что изменение версии правила сортировки может повлиять на порядок сортировки ваших данных, перестройте объекты, использующие правило сортировки по умолчанию. Чтобы убрать эти предупреждения, вы можете выполнить команду `ALTER COLLATION "default" REFRESH VERSION`, как описано в ALTER COLLATION.

Так как в Postgres Pro Standard 10.7.1 была изменена модель распространения pg_probackup, при переходе с предыдущей версии в системе на базе Debian и Альт Линукс выполните `apt dist-upgrade` (или `apt-get dist-upgrade`), чтобы корректно обработались все новые зависимости. В Windows для завершения обновления вам нужно будет запустить отдельный инсталлятор pg_probackup.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям по миграции на версию 10. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.13. Postgres Pro Standard 10.11.1

Дата выпуска: 2019-12-09

E.13.1. Обзор

Этот выпуск основан на PostgreSQL 10.11 и Postgres Pro Standard 10.10.2. Все усовершенствования, унаследованные от PostgreSQL 10.11, перечислены в Замечаниях к выпуску PostgreSQL 10.11. Другие важные изменения и улучшения перечислены ниже:

- Добавлена поддержка ОС Альт Линукс 9.
- Добавлена поддержка системы Ubuntu 19.10.
- Прекращена поддержка системы Ubuntu 18.10.
- Изменён порядок работы с действующими по умолчанию правилами сортировки ICU, чтобы сервер мог запускаться после обновления версии библиотеки ICU. Подробнее следствия изменения описаны в нижеследующих инструкциях по миграции.
- Снижено потребление памяти при использовании правил сортировки ICU.
- Уменьшено потребление памяти и увеличена скорость команды `EXPLAIN ANALYZE` для запросов с множеством предложений `JOIN`.
- Запрещены обращения к таблицам в иерархии наследования при реализации секционирования одновременно средствами и pg_pathman, и PostgreSQL. Совмещение механизмов секционирования в одной иерархии таблиц не поддерживается, и пытаться реализовать это не следует. Ранее при обращении к таким иерархиям в запросах могли выдаваться повторяющиеся результаты.

- Приложение mamonsu обновлено до версии 2.4.1. С новой версией вы можете преобразовать определения всех метрик системы и Postgres Pro, поставляемые с mamonsu, в формат, который поддерживает стандартный агент Zabbix.
- pgbouncer обновлён до версии 1.11.0.
- Приложение pg_probackup обновлено до версии 2.2.5.

E.13.2. Миграция на версию 10.11.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Начиная с Postgres Pro Standard 10.11.1, обновление версии ICU не препятствует запуску сервера. Перед подключением к базе данных, правило сортировки по умолчанию в которой использует ICU, Postgres Pro сравнивает версию этого правила сортировки с версией, предоставляемой библиотекой ICU, и выдаёт предупреждения в случае несовпадения версий; если вы полагаете, что изменение версии правила сортировки может повлиять на порядок сортировки ваших данных, перестройте объекты, использующие правило сортировки по умолчанию. Чтобы убрать эти предупреждения, вы можете выполнить команду `ALTER COLLATION "default" REFRESH VERSION`, как описано в ALTER COLLATION.

Так как в Postgres Pro Standard 10.7.1 была изменена модель распространения pg_probackup, при переходе с предыдущей версии в системе на базе Debian и Альт Линукс выполните `apt dist-upgrade` (или `apt-get dist-upgrade`), чтобы корректно обработались все новые зависимости. В Windows для завершения обновления вам нужно будет запустить отдельный инсталлятор pg_probackup.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям по миграции на версию 10. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.14. Postgres Pro Standard 10.10.2

Дата выпуска: 2019-10-09

E.14.1. Обзор

Этот выпуск основан на Postgres Pro Standard 10.10.1 и включает следующие усовершенствования:

- Добавлена поддержка ОС CentOS 8.
- Функция сравнения выражений была переработана с целью исправления ошибок и улучшения поддержки в будущем.

E.14.2. Миграция на версию 10.10.2

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Так как в Postgres Pro Standard 10.7.1 была изменена модель распространения pg_probackup, при переходе с предыдущей версии в системе на базе Debian и Альт Линукс выполните `apt`

`dist-upgrade` (или `apt-get dist-upgrade`), чтобы корректно обработались все новые зависимости. В Windows для завершения обновления вам нужно будет запустить отдельный инсталлятор pg_probackup.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям в Замечаниях к выпуску Postgres Pro Standard 10.1.1. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.15. Postgres Pro Standard 10.10.1

Дата выпуска: 2019-08-12

E.15.1. Обзор

Этот выпуск основан на PostgreSQL 10.10 и Postgres Pro Standard 10.9.1. Все усовершенствования, унаследованные от PostgreSQL 10.10, перечислены в Замечаниях к выпуску PostgreSQL 10.10. Другие важные изменения и улучшения перечислены ниже:

- Добавлена поддержка ОС Debian 10.
- Добавлена поддержка Red Hat Enterprise Linux 8.
- Улучшена точность планирования запросов с предложениями `OR`. Теперь результаты таких запросов сортируются корректно.
- Исправлена реализация операторов > (больше) и <> (не равно) для типа `jsquery`.

E.15.2. Миграция на версию 10.10.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Так как в Postgres Pro Standard 10.7.1 была изменена модель распространения pg_probackup, при переходе с предыдущей версии в системе на базе Debian и Альт Линукс выполните `apt dist-upgrade` (или `apt-get dist-upgrade`), чтобы корректно обработались все новые зависимости. В Windows для завершения обновления вам нужно будет запустить отдельный инсталлятор pg_probackup.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям в Замечаниях к выпуску Postgres Pro Standard 10.1.1. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.16. Postgres Pro Standard 10.9.1

Дата выпуска: 2019-06-26

E.16.1. Обзор

Этот выпуск основан на PostgreSQL 10.9 и Postgres Pro Standard 10.7.1. Все усовершенствования, унаследованные от PostgreSQL 10.9, перечислены в Замечаниях к выпуску PostgreSQL 10.9. Другие важные изменения и улучшения перечислены ниже:

- Добавлена поддержка ОС Rosa Enterprise Linux Server 7 и РОСА «КОБАЛЬТ» (серверная редакция) на платформе РОСА 7. Предыдущие версии этих операционных систем более не поддерживаются.
- Добавлена поддержка ОС SUSE Linux Enterprise Server 12 SP3 и SUSE Linux Enterprise Server 15.
- Добавлена поддержка ОС Ubuntu 19.04; поддержка Ubuntu 14.04 прекращена.
- Добавлена поддержка ОС AlterOS 7.5.
- В Windows устранена проблема с записью временных файлов статистики в момент, когда они используются.
- Модуль `pg_probackup` обновлён до версии 2.1.3. От версии 2.0.26 её отличают прежде всего следующие улучшения:
  - Резервное копирование и восстановление удалённого экземпляра Postgres Pro по SSH.
  - Объединение инкрементальных копий с их родительскими копиями, которые должны быть удалены согласно политике сохранения.
  - Резервное копирование файлов и каталогов, расположенных вне каталога данных Postgres Pro, например файлов конфигурации или журналов сервера.
  - Команда `checkdb`, предназначенная для проверки всех файлов данных экземпляра Postgres Pro и для логической проверки индексов с использованием amcheck.

  Полный список изменений можно найти на *вики-странице pg_probackup*.
- Модуль `pg_pathman` обновлён до версии 1.5.8. Эта версия по сравнению с 1.5.5, поставляемой в предыдущем выпуске Postgres Pro, включает следующие усовершенствования:
  - Исправлена работа с таблицами, разбитыми на многоуровневые секции. Ранее команды `SELECT FOR SHARE` и `SELECT FOR UPDATE` для таких таблиц могли выдавать следующую ошибку: `ERROR: variable not found in subplan target lists` (ОШИБКА: переменная не найдена в целевых списках подплана).
  - Повышена стабильность `pg_pathman`.

E.16.2. Миграция на версию 10.9.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Так как в Postgres Pro Standard 10.7.1 была изменена модель распространения pg_probackup, при переходе с предыдущей версии в системе на базе Debian и Альт Линукс выполните `apt dist-upgrade` (или `apt-get dist-upgrade`), чтобы корректно обработались все новые зависимости. В Windows для завершения обновления вам нужно будет запустить отдельный инсталлятор pg_probackup.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям в Замечаниях к выпуску Postgres Pro

Standard 10.1.1. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.17. Postgres Pro Standard 10.7.1

Дата выпуска: 2019-03-27

E.17.1. Обзор

Этот выпуск основан на PostgreSQL 10.7 и Postgres Pro Standard 10.6.1. Все усовершенствования, унаследованные от PostgreSQL 10.7, перечислены в Замечаниях к выпуску PostgreSQL 10.7. Другие важные изменения и улучшения перечислены ниже:

- Исправлена ошибка при обработке запросов с множеством предложений `OR`, приводившая к дублированию результатов и нарушению порядка сортировки.
- Устранён дефект, провоцирующий сбой сервера при использовании нестрогих агрегатных функций.
- Изменена модель распространения утилиты pg_probackup:
  - В Linux программа pg_probackup теперь предоставляется в пакете `pg-probackup-std-10`. В Альт Линукс и ОС на базе Debian для обновления Postgres Pro Standard с версии 10.6.1 или ниже выполните `apt dist-upgrade` (или `apt-get dist-upgrade`), чтобы корректно обработались все новые зависимости пакетов.
  - Для Windows теперь предоставляется отдельный инсталлятор pg_probackup. Прежде чем устанавливать pg_probackup, вы должны установить основные компоненты текущей версии Postgres Pro.
- В интерактивном инсталляторе Postgres Pro для Windows появились новые возможности. Теперь вы можете:
  - Выбрать в качестве провайдера основного правила сортировки `icu` или `libc`. Ранее в отсутствие такого выбора в Postgres Pro Standard всегда выбирался провайдер `icu`.
  - Выбрать пользователя, который будет запускать службу Postgres Pro. По умолчанию служба Postgres Pro запускается от имени пользователя `NT AUTHORITY\NetworkService`.
- В SUSE Linux Enterprise Server и системах на базе RHEL несколько утилит перемещены в другие пакеты для согласованности с другими дистрибутивами Linux: pg_receivewal и pg_recvlogical теперь содержатся в пакете `postgrespro-std-10-client`, а pg_rewind, pg_waldump и pg_upgrade — в пакете `postgrespro-std-10-server`.
- Интегрировано исправление, позволяющее избежать условий гонки в процессе `dsm_attach()` при повторном использовании указателей DSM; в ванильном PostgreSQL это исправление запланировано только в будущих выпусках. Ожидается, что в результате разрешатся проблемы, при которых сервер выдаёт следующее сообщение об ошибке: `ERROR: dsa_area could not attach to segment` (ОШИБКА: dsa_area не может получить доступ к сегменту).
- Исправлена работа pg_upgrade с ограничениями доменов, зависящими от правил сортировки. Такие ограничения теперь проверяются в процессе обновления, тогда как в предыдущих версиях Postgres Pro они всегда помечались в новом кластере как непроверенные.
- Исправлен скрипт pg-setup; теперь с ключом `-D` он позволяет инициализировать кластер в нестандартном расположении.
- Утилита pg_probackup теперь корректно отслеживает в режиме `PTRACK` изменения в индексах `rum` и `bloom`.
- Устранено снижение производительности при сканировании только индекса, когда индекс построен по множеству полей.
- Прекращена поддержка систем Альт Линукс СПТ 6.0 и Windows 7 SP1.

- Модуль pg_pathman обновлён до версии 1.5.5. Эта версия по сравнению с 1.5.2, поставляемой в предыдущем выпуске Postgres Pro, включает следующие усовершенствования:
  - Усовершенствованы скрипты обновления pg_pathman во избежание ошибок при миграции с основной версии 1.4 на 1.5 из-за различного числа атрибутов pg_config в этих версиях.
  - Повышена стабильность pg_pathman:
    - Попытки вызова функций pg_pathman при отключённом расширении теперь не будут приводить к сбоям сервера; вместо этого будут выдаваться ошибки.
    - К одной и той же таблице могут быть применены различные стратегии секционирования.
- Модуль pg_variables обновлён до версии 1.2:
  - Добавлена поддержка переменных-массивов.
  - Повышена стабильность модуля.
  - Исправлено удаление пустых пакетов. Теперь пустой пакет будет удаляться только после завершения транзакции, в которой он был опустошён.

  (См. Раздел F.44.)

E.17.2. Миграция на версию 10.7.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Так как в Postgres Pro Standard 10.7.1 была изменена модель распространения pg_probackup, при переходе с предыдущей версии в системе на базе Debian и Альт Линукс выполните apt dist-upgrade (или apt-get dist-upgrade), чтобы корректно обработались все новые зависимости. В Windows для завершения обновления вам нужно будет запустить отдельный инсталлятор pg_probackup.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду REINDEX для индексов, в которых используются типы mchar или mvarchar. Кроме этого, если вы использовали pg_repack в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в pg-repack-std-10.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям в Замечаниях к выпуску Postgres Pro Standard 10.1.1. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр --add-collprovider, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.18. Postgres Pro Standard 10.6.1

Дата выпуска: 2018-11-20

E.18.1. Обзор

Этот выпуск основан на PostgreSQL 10.6 и Postgres Pro Standard 10.5.3. Все усовершенствования, унаследованные от PostgreSQL 10.6, перечислены в Замечаниях к выпуску PostgreSQL 10.6. Другие важные изменения и улучшения перечислены ниже:

- Добавлена поддержка операционных систем Ubuntu 18.10, Astra Linux «Смоленск» 1.6 и РЕД ОС 7 МУРОМ.
- Усовершенствован скрипт pg-setup:
  - Теперь он может инициализировать кластер баз данных в нестандартном расположении и сохранить соответствующее значение PGDATA в системном конфигурационном файле.

    - Запустив pg-setup с ключом `set`, теперь можно изменить конфигурацию кластера.
- Добавлена возможность замены нулевого байта заданным ASCII-символом при загрузке данных с помощью команды `COPY FROM`. Подробнее об этом можно узнать в описании параметра nul_byte_replacement_on_import.
- Расширение `amcheck` обновлено до версии 1.1.
- Улучшена стабильность plantuner и устранена утечка памяти.
- Улучшено планирование запросов со множеством операторов `OR` в предложении `WHERE`.
- Модуль `pg_pathman` обновлён до версии 1.5.2. Эта версия по сравнению с 1.4.14, входившей в предыдущий выпуск Postgres Pro, включает следующие усовершенствования:
    - Добавлена поддержка многоуровневого секционирования.
    - Ликвидированы триггеры на изменение и добавлен параметр `pg_pathman.enable_partitionrouter`, включающий межсекционные операции изменений.
    - Функция `get_pathman_lib_version()` переименована в `pathman_version()`.
    - В новую версию вошли и другие улучшения и исправления ошибок. Полный список изменений можно найти на *вики-странице pg_pathman*.
- Модуль `pg_probackup` обновлён до версии 2.0.24. Эта версия по сравнению с 2.0.19, поставляемой в предыдущем выпуске Postgres Pro, отличается следующими усовершенствованиями:
    - Файлы, которые не содержат данных и не изменились со времени предыдущего копирования, не включаются в инкрементальную копию.
    - Номер версии, заданный в `pg_probackup.conf`, теперь сохраняется при изменении этого файла, что позволяет точно узнать, какая версия pg_probackup применялась при создании резервной копии.
    - Устранена проблема восстановления сжатых блоков файлов и улучшены проверки ошибок сжатия. Ранее pg_probackup не мог восстановить блоки файлов, которые алгоритм `zlib` не смог сжать во время резервного копирования. Эту проблему нельзя было выявить встроенным механизмом проверки pg_probackup, так как она проявлялась на уровне ниже, чем уровень проверки. В сложившейся ситуации рекомендуется перепроверить существующие резервные копии, используя эту версию pg_probackup.
    - Улучшен механизм проверки целостности. Файлы теперь по умолчанию проверяются поблочно не только в случае несовпадения контрольной суммы на уровне файлов. Это поведение можно отключить с помощью параметра `--skip-block-validation`.
    - Появилась возможность перезапустить объединение резервных копий, если предыдущая попытка была прервана.
    - Реализована возможность снятия резервных копий с ведомых серверов без подключения к ведущему. Также в pg_probackup теперь появился встроенный механизм определения точки согласованности, исключающий риски несогласованности данных в копиях, сделанных на ведомых серверах.
- Версию Postgres Pro для Windows затронули следующие изменения:
    - Для PL/Perl теперь требуется ActivePerl 5.26.
    - 32-битные версии Postgres Pro более не выпускаются.

E.18.2. Миграция на версию 10.6.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям в Замечаниях к выпуску Postgres Pro Standard 10.1.1. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.19. Postgres Pro Standard 10.5.3

Дата выпуска: 2018-10-11

E.19.1. Обзор

Этот выпуск основан на Postgres Pro Standard 10.5.2 и включает следующие усовершенствования:

- Исправлена ошибка в вычислении минимальной точки восстановления на резервных серверах, которая могла повлечь обращение к неправильным страницам.
- Обновлён модуль online_analyze; теперь он не допускает вложенные вызовы `ANALYZE`.
- Число разделов хеш-таблицы сопоставлений в общем буфере увеличено до 1024, что может положительно сказаться на производительности в многоядерных системах.
- Устранена ошибка восстановления резервной копии на ведущем сервере, возникавшая в условиях гонки при применении двухфазных транзакций.
- В системах Windows устранена проблема с перезагрузкой словарей, предоставляемых модулем `shared_ispell`.
- Утилита pg_probackup обновлена до версии 2.0.21, в которой представлены следующие усовершенствования:
  - Устранены ошибки, связанные с восстановлением резервных копий, сделанных на ведомых серверах.
  - Исправлен разбор параметров `log-rotation-size` и `log-rotation-age`.
  - Команда `show` теперь динамически меняет ширину выводимых строк для улучшения читабельности больших таблиц.
  - Команда `restore` теперь корректно восстанавливает все символические ссылки на табличные пространства.
  - Если включены контрольные суммы, команда `validate` теперь проверяет контрольные суммы блоков.

E.19.2. Миграция на версию 10.5.3

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям в Замечаниях к выпуску Postgres Pro

Standard 10.1.1. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.20. Postgres Pro Standard 10.5.2

Дата выпуска: 2018-09-17

E.20.1. Обзор

Этот выпуск основан на Postgres Pro Standard 10.5.1 и включает следующие исправления:

- Модуль `pg_pathman` обновлён до версии 1.4.14 в целях повышения стабильности работы.

E.20.2. Миграция на версию 10.5.2

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям в Замечаниях к выпуску Postgres Pro Standard 10.1.1. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.21. Postgres Pro Standard 10.5.1

Дата выпуска: 2018-08-20

E.21.1. Обзор

Этот выпуск основан на PostgreSQL 10.5 и Postgres Pro Standard 10.4.1. Все усовершенствования, унаследованные от PostgreSQL 10.5, перечислены в Замечаниях к выпуску PostgreSQL 10.5. Другие важные изменения и улучшения перечислены ниже:

- Модуль `pg_variables` теперь поддерживает транзакционные переменные. (См. Раздел F.44.)
- Модуль `auto_explain` теперь может выводить время планирования.
- Исправлена ошибка, из-за которой нельзя было пользоваться некоторыми словарями Hunspell с аффиксами `FLAG num`, например, `ru_aot`.
- Модуль pg_probackup обновлён до версии 2.0.19, в которой появились новые возможности:
  - Теперь кластер можно восстановить до заданной позиции в журнале (LSN).
  - Если одна из родительских копий испорчена, данная инкрементальная копия получает состояние `ORPHAN` (потерянная).
  - Команда `show-config` теперь показывает и изменённые параметры pg_probackup, и значения, оставшиеся неизменными. Её вывод теперь можно получить в формате JSON для упрощения восприятия.
  - Команда `restore` может пропускать проверку копии для ускорения восстановления кластера.

- Улучшено параллельное выполнение инкрементального копирования.
- Теперь вы можете объединять инкрементальные копии с родительской полной копией для экономии места на диске. Будьте осторожны, это экспериментальная возможность — прерывание процесса объединения может привести к повреждению резервных копий.

- Усовершенствована версия Postgres Pro Standard для Windows:
  - Устранено неявное ограничение на количество одновременно открываемых файлов в каждом серверном подпроцессе.
  - Добавлена возможность отключения контрольных сумм в кластере. По умолчанию при установке Postgres Pro контрольные суммы включаются.
  - Добавлена поддержка модуля pg_probackup.

E.21.2. Миграция на версию 10.5.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`. Кроме этого, если вы использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям в Замечаниях к выпуску Postgres Pro Standard 10.1.1. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.22. Postgres Pro Standard 10.4.1

Дата выпуска: 2018-05-23

E.22.1. Обзор

Этот выпуск основан на PostgreSQL 10.4 и Postgres Pro Standard 10.3.2. Все усовершенствования, унаследованные от PostgreSQL 10.4, перечислены в Замечаниях к выпуску PostgreSQL 10.4. Другие важные изменения и улучшения перечислены ниже:

- Доработано расширение Раздел F.29 для правильной сортировки букв «й» и «ё». Перестройте индексы, использующие типы `mchar` или `mvarchar`, с помощью команды `REINDEX`.
- В словари Snowball и Hunspell добавлена поддержка полнотекстового поиска для непальского языка. (См. snowball_ext и Модули словарей Hunspell.)
- Обновлён модуль jsquery.
- Доработка PTRACK обновлена до версии 1.6:
  - Теперь ptrack не отслеживает нежурналируемые отношения.
- Утилита pg_probackup обновлена до версии 2.0.17, в которой появились новые возможности:
  - Режим `DELTA`, в котором считываются все файлы данных в каталоге данных и создаётся инкрементальная копия для страниц, изменённых со времени предыдущего копирования.
  - Новые параметры команд `restore` и `validate`:
    - Параметр `--immediate` останавливает восстановление, как только достигается согласованное состояние.

    - Параметр `--recovery-target-action` задаёт действие, которое должен выполнить сервер по достижении цели восстановления.
    - Параметр `--recovery-target-name` указывает именованную точку сохранения, вплоть до которой будут восстановлены данные кластера.
    - С `--write-recovery-conf` в каталог вывода записывается минимальный файл `recovery.conf` для упрощения настройки резервного сервера.

  За подробностями обратитесь к pg_probackup.
- Модуль `pg_pathman` обновлён до версии 1.4.11. Эта версия по сравнению с 1.4.9 включает следующие усовершенствования:
    - Устранена проблема с дублированием записей в результатах запросов с наследованными таблицами. Вообще говоря, многоуровневое секционирование `pg_pathman` не поддерживает.
    - Устранена ложная ошибка `table is being partitioned now` (таблица сейчас секционируется), возникавшая в `partition_table_concurrently()`.
    - Ослаблена обработка ограничения-проверки.
    - Исправлено некорректное использование `memcpy()` в `start_bgworker()`.
    - Полный список изменений можно найти на *вики-странице pg_pathman*.
- В `pg_config` добавлены параметры `pgpro-version` и `pgpro-edition`.
- Файл `recovery.conf` теперь перезагружается, когда главный процесс сервера получает сигнал `SIGHUP`.
- Произведено исправление ряда ошибок в модуле shared_ispell.
- В системах на базе Debian пакет `pg_repack` переименован из `pg-repack-10` в `pg-repack-std-10`.
- Файлы журналов теперь автоматически открываются заново при получении сигнала `SIGHUP` процессом, регистрирующим сообщения.
- Доработана версия Postgres Pro Standard для Windows:
    - Устранена возникавшая в Windows проблема при вставке из буфера обмена в psql строк, содержащих знаки из разных наборов символов.
    - Усовершенствован алгоритм оптимизации предлагаемой по умолчанию конфигурации.
    - Теперь, устанавливая Postgres Pro в командной строке, вы можете отключить оптимизацию конфигурации, присвоив в INI-файле параметру `needoptimization` значение `0`.

E.22.2. Миграция на версию 10.4.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

> **Важно**
>
> При обновлении с версии 10.3.2 и ниже вы должны выполнить команду `REINDEX` для индексов, в которых используются типы `mchar` или `mvarchar`.

> **Важно**
>
> Если вы ранее использовали `pg_repack` в системах на базе Debian, при переходе на эту версию вы должны будете переустановить соответствующий пакет вручную, так как он был переименован в `pg-repack-std-10`.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям в Замечаниях к выпуску Postgres Pro Standard 10.1.1. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.23. Postgres Pro Standard 10.3.2

Дата выпуска: 2018-03-20

E.23.1. Обзор

Этот выпуск основан на PostgreSQL 10.3 и Postgres Pro Standard 10.3.1. Все усовершенствования, унаследованные от PostgreSQL 10.3, перечислены в Замечаниях к выпуску PostgreSQL 10.3. Другие важные изменения и улучшения перечислены ниже:

- Добавлена поддержка TOAST для атрибутов INCLUDED в индексах-B-деревьях.
- Исправлен поиск типов `mchar` и `mvarchar` через `search_path`. Теперь эти типы могут быть выгружены даже с очищенным путём поиска.
- Добавлен ключ `--add-collprovider` в утилиты `pg_dump` и `pg_dumpall`.
- Утилита pg_probackup обновлена с версии 2.0.14 до 2.0.16, в которой появились новые возможности:
  - Устранён бесконечный цикл, который был возможен при проверке страниц.
  - Устранён критический сбой в случае параллельных подключений `PTRACK`.
  - Параметр `delete-wal` теперь можно использовать без `delete-expired`.
  - Имя схемы теперь явно указывается во всех вызовах функций для устранения уязвимости CVE-2018-1058.
  - Файлы данных в каталоге `global` теперь обрабатываются корректно: pg_probackup может делать инкрементальные копии и производить проверку контрольных сумм для таких файлов.
  - Файлы `PTRACK` из каталога `global` более не включаются в резервную копию.
  - pg_probackup теперь учитывает аргумент `--instance` команды `VALIDATE`.
- Исправлена проблема с сортировкой по порядку некоторых русских букв в модуле `mchar`. Для обновления индексов воспользуйтесь командой `REINDEX DATABASE`.

E.23.2. Миграция на версию 10.3.2

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям в Замечаниях к выпуску Postgres Pro Standard 10.1.1. Если вы выбираете вариант с выгрузкой/восстановлением данных, обязательно используйте параметр `--add-collprovider`, чтобы в восстановленной базе данных оказался корректный провайдер основного правила сортировки.

E.24. Postgres Pro Standard 10.3.1

Дата выпуска: 2018-03-01

E.24.1. Обзор

Этот выпуск основан на PostgreSQL 10.3 и Postgres Pro Standard 10.2.1. Все усовершенствования, унаследованные от PostgreSQL 10.3, перечислены в Замечаниях к выпуску PostgreSQL 10.3.

E.24.2. Миграция на версию 10.3.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям в Замечаниях к выпуску Postgres Pro 10.1.1.

E.25. Postgres Pro Standard 10.2.1

Дата выпуска: 2018-02-16

E.25.1. Обзор

Этот выпуск основан на PostgreSQL 10.2 и включает все новые возможности, появившиеся в PostgreSQL 10, а также исправления ошибок, вошедшие в PostgreSQL 10.2. За подробным их описанием обратитесь к Замечаниям к выпуску PostgreSQL 10 и Замечаниям к выпуску PostgreSQL 10.2, соответственно. Другие важные изменения и улучшения:

- Каталог по умолчанию для сохранения истории `psql` динамически создаётся в системах Windows.
- Утилита pg_probackup обновлена до версии 2.0.14.
  - Исправлено поведение `ptrack` и резервного копирования страниц в условиях гонки.
  - Исправлено прерывание параллельного резервного копирования `ptrack`.
  - Добавлено новое состояние резервного копирование (`ORPHAN`), указывающее, что одна из предыдущих резервных копий испорчена.
- Обновлён модуль pg_variables.
- Модуль `pg_pathman` обновлён до версии 1.4.9. В этой версии исправлена обработка `ONLY` во всех типах запросов. (См. pg_pathman и Подраздел F.36.1.1.)
- Модуль dump_stat обновлён до версии 1.1, включающей ряд исправлений различных ошибок. (См. dump_stat.)
- Исправлена некорректная обработка условий `IS NULL` и `IS NOT NULL` в индексах с предложениями `INCLUDE`.
- Добавлена правка для совместимости `int2vector`.
- Обновлён модуль online-analyze.

E.25.2. Миграция на версию 10.2.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям в Замечаниях к выпуску Postgres Pro 10.1.1.

E.26. Postgres Pro Standard 10.1.1

Дата выпуска: 2017-12-14

E.26.1. Обзор

Этот выпуск основан на PostgreSQL 10.1 и включает все новые возможности, появившиеся в PostgreSQL 10, а также исправления ошибок, вошедшие в PostgreSQL 10.1. За подробным их описанием обратитесь к Замечаниям к выпуску PostgreSQL 10 и Замечаниям к выпуску PostgreSQL 10.1, соответственно. Другие важные изменения и улучшения:

- Унифицирована структура пакетов двоичной установки для разных дистрибутивов Linux. Новая структура пакетов отличается от структуры ванильного PostgreSQL, но даёт следующие преимущества:
  - Вам не нужно беспокоиться о специфике инсталляций в разных дистрибутивах Linux при миграции между различными поддерживаемыми системами Linux. Postgres Pro Standard теперь устанавливается в каталог `/opt/pgpro/std-10`, а база данных по умолчанию создаётся в каталоге `/var/lib/pgpro/std-10/data`.
  - Теперь вы можете полностью управлять созданием базы данных по умолчанию во всех дистрибутивах Linux. Если вы устанавливаете пакет `postgrespro-std-10`, он разворачивает все пакеты Postgres Pro, необходимые для вашей системы, создаёт базу данных по умолчанию и настраивает сервер полностью автоматическим образом. Если вы устанавливаете отдельные пакеты, вам нужно настроить Postgres Pro самостоятельно. В этом случае вы должны вручную инициализировать кластер баз данных и запустить сервер, а также настроить автозапуск сервера, если требуется.
  - Вы можете установить Postgres Pro Standard 10 рядом с другими продуктами на базе PostgreSQL для осуществления миграции или параллельного использования. Если вы выбираете отдельные пакеты, ваша системная конфигурация будет сохранена, так что вам придётся вручную настроить Postgres Pro, следуя инструкциям в Подразделе 16.1.3. Не устанавливайте пакет `postgrespro-std-10` в системе, где уже установлен другой продукт на базе PostgreSQL, во избежание конфликтов.

  Заметьте, что пакеты `postgrespro-common` и `postgrespro-client-common` для систем на базе Debian теперь отсутствуют. За подробностями обратитесь к Разделу 16.1.
- Возможность установки `icu` или `libc` в качестве провайдера основного правила сортировки при инициализации кластера баз данных или создании базы данных. По умолчанию для всех локалей, кроме `C` и `POSIX`, используется провайдер `icu`. За подробностями обратитесь к Подразделу 22.2.2.
- В индексе-B-дереве дублирующиеся ключи теперь хранятся в единственном экземпляре, вместо дубликатов хранятся указатели на элементы в списке. Это может значительно сократить размер индекса, когда один ключ встречается многократно.

Ключевые доработки, перенесённые из Postgres Pro Standard 9.6:

- Доработки покрывающих индексов (За подробностями обратитесь к описанию предложения `INCLUDE` в CREATE INDEX.)
- Исправления для системы сборки win32
- Добавлена SQL-функция pgpro_version и соответствующие определения в pg_config.h
- Внедрение механизма PTRACK
- В psql для Windows включена поддержка редактирования командной строки с использованием `libeditline`

Из Postgres Pro Standard 9.6 перенесены следующие модули и утилиты:

- hunspell-dict (см. Модули словарей Hunspell)
- jsquery (см. Раздел F.26)
- pg_variables (см. Раздел F.44)
- pg_pathman (см. pg_pathman)
- pg_query_state (см. Раздел F.38)
- shared-ispell (см. shared_ispell)
- sr_plan (см. sr_plan)
- dump_stat (см. dump_stat)
- pg_buffercache (см. pg_buffercache)
- mchar (см. Раздел F.29)
- plantuner (см. Раздел F.46)
- fasttrun (см. Раздел F.17)
- fulleq (см. Раздел F.19)
- online_analyze (см. Раздел F.30)

- pg_pathman (см. pg_pathman)
- утилита pg_probackup (см. pg_probackup)

E.26.2. Миграция на версию 10

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, сначала установите его последний корректирующий выпуск, а затем выполните выгрузку/восстановление данных, применив pg_dumpall, или воспользуйтесь pg_upgrade. Если вы решите использовать pg_upgrade, убедитесь в том, что новый кластер баз данных имеет ту же характеристику контрольных сумм, что и кластер, из которого вы переносите данные.

При миграции на Postgres Pro Standard 10 во избежание конфликтов используйте не пакет `postgrespro-std-10`, а отдельные пакеты. В этом случае автозапуск сервера нужно будет настроить вручную, если это потребуется.

Начиная с Postgres Pro Standard версии 10, при инициализации кластера баз данных или при создании базы может быть задан провайдер основного правила сортировки как описано в Подразделе 22.2.2. Вы должны учитывать это при обновлении до данной версии, чтобы избежать повреждения индексов или ограничений.

> **Важно**
>
> Версии PostgreSQL 9.5 и 9.5.1, а также Postgres Pro Standard 9.5.0.1 и 9.5.1.2 нельзя обновить непосредственно до Postgres Pro Standard 10. Если вы используете такую версию, сначала обновите вашу инсталляцию до промежуточной версии, например, до Postgres Pro Standard 9.5.2.1.
>
> Также, если вы используете версию Postgres Pro Standard 9.6.10.x в Windows, сначала обновите её до Postgres Pro Standard версии 9.6.11.1 или выше.

Следуя процедуре обновления с использованием pg_upgrade, убедитесь в том, что initdb устанавливает в новом кластере баз данных корректный провайдер основного правила сортировки:

- Осуществляя миграцию с ванильного сервера PostgreSQL, укажите `libc` в качестве провайдера основного правила сортировки.
- При обновлении Postgres Pro Standard опустите указание провайдера правил сортировки, чтобы этот провайдер был выбран автоматически. В этом случае для баз данных с локалями C и POSIX, а также для баз данных с однобайтовыми кодировками будет использоваться провайдер `libc`, а для всех остальных — провайдер `icu`.

Если pg_upgrade создаёт какие-либо файлы SQL в текущем каталоге, запустите их для завершения обновления.

Когда обновление осуществляется с применением pg_dumpall, Postgres Pro использует провайдер правил сортировки, заданный в команде `initdb` для нового кластера. В этом случае индексы перестраиваются автоматически. Во избежание проблем с ограничениями, зависящими от правил сортировки, рекомендуется использовать провайдер `libc` при обновлении с ванильного PostgreSQL и опустить указание провайдера при обновлении с предыдущей версии Postgres Pro Standard, если только в ваших базах не используются правила сортировки, отличные от C и POSIX. Для таких баз данных выполните следующее:

- Если новый кластер инициализируется с локалью, отличной от C и POSIX, и при этом в базе используется однобайтовая кодировка, установите для этой базы в LC_COLLATE значение '@libc'.
- Если новый кластер инициализируется с локалью C или POSIX, а в базе используется многобайтовая кодировка, установите для этой базы в LC_COLLATE значение `'@icu'`.

> **Примечание**
>
> Если в предыдущей инсталляции Postgres Pro Standard в базах с многобайтовыми кодировками имелись индексы или ограничения, зависящие от правил сортировки, отличных от `C`, `POSIX` или правила сортировки БД по умолчанию, в этих базах могли оказаться данные, нарушающие целостность индексов и условия ограничений. В Windows это также могло наблюдаться, если база данных с многобайтовой кодировкой содержала индексы или ограничения, зависящие от правила сортировки по умолчанию, имеющего длинное имя, например "Russian_Russia[.*кодировка*]" или "English_United States[.*кодировка*]". В таких случаях при обновлении Postgres Pro Standard до версии 10 сценарий с выгрузкой/восстановлением данных может оказаться невозможным и для обновления нужно будет использовать только pg_upgrade. Для устранения описанных проблем pg_upgrade объявляет такие индексы и ограничения нерабочими и создаёт скрипты `reindex_text_indexes.sql` и `validate_text_contraints.sql`, соответственно.

E.27. Выпуск 10.23

Дата выпуска: 2022-11-10

В этот выпуск вошли различные исправления, внесённые после версии 10.22. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

Этот выпуск PostgreSQL должен стать последним в серии 10.X. Пользователям следует поторопиться с переходом на более новую основную версию.

E.27.1. Миграция на версию 10.23

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Если вы обновляете сервер с более ранней версии, чем 10.19, см. также Раздел E.31.

E.27.2. Изменения

- Исправление `VACUUM`, позволяющее продолжать очистку, если при удалении страницы индекса btree в родительской странице не нашлась ссылка на удаляемую страницу (Питер Гейган)

  Теперь в этом случае не выдаётся ошибка, а просто записывается сообщение в журнал и операция продолжается, оставляя пустую страницу. Раньше при наличии дефектного класса операторов или повреждённого индекса очистка индекса не могла завершиться, что в конце концов могло вызвать проблемы с зацикливанием идентификаторов транзакций.

- Исправление обработки элементов `DEFAULT` в предложении `VALUES` с несколькими кортежами при добавлении данных в изменяемое представление (Том Лейн)

  Следствием исправленного теперь упущения могли быть ошибки «cache lookup failed for type» (ошибка поиска в кеше для типа) или даже аварийный сбой в старых версиях.

- Недопущение имени `_RETURN` для всех правил, кроме правил `ON SELECT` (Том Лейн)

  Таким образом, задаваемое для представления правило `ON SELECT` будет отличаться от других задаваемых для него правил.

- Исправление ошибки, редко возникавшей во вложенных планах MULTIEXPR_SUBLINK при `UPDATE` с наследованием (Том Лейн)

  При выполнении команды `UPDATE tab SET (c1, ...) = (SELECT ...)`, где целевая таблица секционирована или имеет потомков, мог произойти сбой, если дочерние таблицы имели существенные различия. Как правило, это проявлялось в нарушении проверок согласованности в исполнителе, но также могла произойти аварийная остановка или данные изменялись некорректно.

- Запрет разворачивания подзапросов без предложения `FROM`, когда во внешнем запросе есть наборы группирования (Том Лейн)

  Раньше из-за такого разворачивания могли возникать сбои проверочных утверждений или ошибки планировщика «variable not found in subplan target list» (переменная не найдена в целевом списке подплана).

- Предотвращение повреждения WAL после повышения ведомого сервера (Дилип Кумар, Роберт Хаас)

  Когда экземпляр PostgreSQL, выполняющий восстановление архива (но не работающий в режиме горячего резерва), переключался в роль ведущего и последний прочитываемый им сегмент WAL завершался неполной записью, экземпляр записывал некорректный сегмент WAL в новой линии времени.

- Исправление неверного порядка операций WAL при добавлении данных в индексы GIN по быстрому пути (Маттиас ван де Меент, Мингли Чжан)

  О негативном влиянии этого дефекта на работу ядра PostgreSQL неизвестно, но с некоторыми расширениями возникали проблемы.

- Исправление ошибок в логическом декодировании при запуске воспроизведения с позиции между началом транзакции и началом подтранзакции (Масахико Савада, Хайато Курода)

  Устранённые теперь ошибки могли приводить к сбоям проверочных утверждений в отладочных сборках или же к утечкам памяти.

- Предотвращение обращения к системным каталогам с использованием неверного снимка во время логического декодирования (Масахико Савада)

  Если декодирование начиналось посреди транзакции, изменяющей системные каталоги, этот факт мог остаться незамеченным, и поиск по каталогу при декодировании производился без учёта данной транзакции.

- Добавление точек, в которых возможно прерывание при логическом декодировании (Амит Капила, Масахико Савада)

  Тем самым устраняются проблемы медленного отключения рабочих процессов репликации.

- Устранение сбоя рабочего процесса репликации в случае синтаксической ошибки в функции (Максим Орлов, Антон Мельников, Масахико Савада, Том Лейн)

  Если в команде `CREATE FUNCTION` или `DO` на языке SQL или PL/pgSQL была допущена синтаксическая ошибка, при выполнении этой команды рабочим процессом репликации он завершался аварийно из-за обращения по нулевому указателю или сбоя проверочного утверждения.

- Исправление обработки допускающих изменение развёрнутых данных, передаваемых в функции SQL (Том Лейн)

  В случае неоднократного использования одного параметра в невстроенной SQL-функции, когда предполагается, что вызываемая из неё SQL-функция должна модифицировать его данные на месте, при следующих обращениях к этому параметру его значение было некорректным. (В ядре PostgreSQL механизм развёрнутых данных используется только для массивов и значений составного типа; однако расширения могут использовать его и для других структурированных типов.)

- Отказ от вызова стеммера в Snowball для чрезмерно длинных слов (Олли Беттс, Том Лейн)

  Если размер слова превышает 1000 байт, оно не обрабатывается кодом Snowball, а возвращается как есть после преобразования регистра. Это ограничение помогает избежать известной проблемы переполнения стека при рекурсии в стеммере для турецкого языка, а также представляется хорошей защитой от других проблем с безопасностью или производительностью, возможных в стеммерах Snowball. Строки такого размера не могут

быть словами какого-либо человеческого языка, поэтому сомнительно, что стеммер мог бы выдать желаемый результат в любом случае.

- Ликвидация риска использования освобождённой памяти при сравнении строк (Том Лейн)

  Вследствие некорректного управления памятью в функциях сравнения строк код мог писать в уже освобождённые буферы и тем самым навредить следующему пользователю этой памяти. Это могло происходить при сравнении только довольно длинных строк (больше 1 КБ) и только при использовании правила сортировки ICU.

- Предотвращение краха процесса postmaster при повреждении общей памяти (Том Лейн)

  Предполагается, что повреждение общей памяти не должно быть фатальным для процесса postmaster, он должен штатно перезапустить базу данных, но в одном месте кода были предприняты не все меры для этого.

- Добавление дополнительной защиты от переполнения стека при рекурсии (Ричард Гуо, Том Лейн)

- Предотвращение долгосрочной утечки памяти в процессе запуска автоочистки (Рейд Томпсон)

  Сообщения об этой проблеме не поступали, что говорит о том, что до версии 15 она была скрытой, не вполне понятно почему; в любом случае исправление перенесено и в старые версии.

- Добавление в libpq недостающих проверок указателя соединения на `NULL` (Даниэле Вараццо, Том Лейн)

  Есть соглашение, что функции libpq должны проверять, не передан ли NULL в аргументе PGconn, и в этом случае завершаться штатной ошибкой, а не крахом. Функции `PQflush()` и `PQisnonblocking()` этому соглашению не следовали, но теперь они исправлены.

- Устранение в ecpg исчезания класса хранения переменных при описании в одном объявлении нескольких переменных `varchar` или `bytea` (Андрей Соколов)

  Например, объявление `static varchar str1[10], str2[20], str3[30];` преобразовывалось в ecpg таким образом, что пометка `static` оставалась только у переменной `str1`.

- Обеспечение возможности кроссплатформенного перемещения табличных пространств в pg_basebackup (Том Лейн)

  Теперь удалённый путь в `--tablespace-mapping` может задаваться как абсолютный путь и в стиле Unix, и в стиле Windows, поскольку ОС исходного сервера и локальной системы могут различаться.

- Устранение обращения к освобождённой памяти в pg_stat_statements (Цигуй Чжао)

  Проблема возникала, когда в pg_stat_statements отслеживалась команда `ROLLBACK`, переданная по протоколу расширенных запросов. В отладочных сборках при этом гарантированно происходили сбои проверочных утверждений, а в обычных чаще всего никаких заметных негативных последствий не было, но если освобождённая память использовалась снова, это было чревато сохранением мусора в строке запроса.

- Обеспечение в postgres_fdw наличия всех необходимых столбцов в целевых списках, формируемых для планов EvalPlanQual (Ричард Гуо, Эцуро Фудзита)

  Тем самым предупреждаются ошибки «variable not found in subplan target list» (переменная не найдена в целевом списке подплана), возникавшие в редких случаях.

- Недопущение нежелательного вывода системной функции `uuid_create()` (Назир Билал Явуз)

  Модуль uuid-ossp ожидает, что функция libc `uuid_create()` вернёт UUID версии 1, но в последних версиях NetBSD возвращается UUID версии 4 (случайный). Теперь версия проверяется, и в случае несоответствия выдаётся ошибка. Из документации удалено

упоминание NetBSD в перечне систем, подходящих для использования uuid-ossp. (Если для ваших целей подходят UUID версии 4, uuid-ossp вам вовсе не нужен — используйте функцию `gen_random_uuid()`.)

- Включение новых модулей Perl для тестирования в стандартную инсталляцию (Альваро Эррера)

  `PostgreSQL/Test/Cluster.pm` и `PostgreSQL/Test/Utils.pm` добавлены в список файлов для стандартной инсталляции версий до 15. Это может быть полезно для тестирования расширений, когда возникает желание использовать новые тесты в старых версиях.

- Обеспечение в NetBSD динамического разрешения символов в момент запуска процесса postmaster (Андрес Фройнд, Том Лейн)

  Тем самым устраняется риск взаимоблокировки при выполнении динамического связывания в NetBSD 10.

- Добавление возможности использовать `__sync_lock_test_and_set()` для циклических блокировок на любой машине (Том Лейн)

  Это упрощает перенос на новую машинную архитектуру, по крайней мере, если вы используете компилятор, поддерживающий эту встроенную в GCC функцию.

- Переименование символа `REF` в `REF_P` во избежание проблем при компиляции в последних версиях macOS (Том Лейн)

- Устранение различных предупреждений, выдаваемых компилятором clang версии 15 и выше (Том Лейн)

- Обновление данных часовых поясов до версии tzdata 2022f, включающее изменение правил перехода на летнее время в Чили, Иране, Иордании, Мексике, Палестине, Сирии и на Фиджи, а также корректировку исторических данных для Чили, Крыма, Ирана и Мексики.

  Кроме того, пояс Europe/Kiev был переименован в Europe/Kyiv. Помимо этого, следующие пояса были включены в соседние, более популярные пояса, в которых время было таким же с 1970 г.: Antarctica/Vostok, Asia/Brunei, Asia/Kuala_Lumpur, Atlantic/Reykjavik, Europe/Amsterdam, Europe/Copenhagen, Europe/Luxembourg, Europe/Monaco, Europe/Oslo, Europe/Stockholm, Indian/Christmas, Indian/Cocos, Indian/Kerguelen, Indian/Mahe, Indian/Reunion, Pacific/Chuuk, Pacific/Funafuti, Pacific/Majuro, Pacific/Pohnpei, Pacific/Wake и Pacific/Wallis. (Это косвенно влияет на пояса, ранее включённые в другие: Arctic/Longyearbyen, Atlantic/Jan_Mayen, Iceland, Pacific/Ponape, Pacific/Truk и Pacific/Yap.) Пояса America/Nipigon, America/Rainy_River, America/Thunder_Bay, Europe/Uzhgorod и Europe/Zaporozhye также были включены в соседние, поскольку выяснилось, что заявления об отличиях этих поясов от соседних (после 1970 года) оказались ошибочными. Для всех этих поясов старое название сохранено в качестве альтернативного; фактически используются данные пояса, в который они были включены.

  В результате такого включения для включённых поясов теряется история до 1970 г., что может вызвать проблемы в приложениях, рассчитывающих на согласованный вывод `timestamptz`. Например, хранимое значение `1944-06-01 12:00 UTC` ранее для часового пояса Europe/Stockholm выводилось как `1944-06-01 13:00:00+01`, а теперь будет выводиться как `1944-06-01 14:00:00+02`.

  В принципе возможно собрать файлы часовых поясов так, чтобы были восстановлены данные старых поясов, но при этом будут добавлены и другие старые (и, как правило, плохо проверенные) данные поясов, что принесёт по сравнению с предыдущей версией больше изменений, чем просто принятие изменений из проекта tzdata. В PostgreSQL было решено включить в поставку рекомендованные данные tzdb, и насколько нам известно, производители большинства операционных системы поступают так же. Тем не менее, если эти изменения приводят к существенным проблемам в вашем приложении, возможным решением будет установка локальной сборки файлов с данными часовых поясов tzdb, полученной с указанием параметров для обратной совместимости (см. параметры `PACKRATDATA` и `PACKRATLIST`).

E.28. Выпуск 10.22

Дата выпуска: 2022-08-11

В этот выпуск вошли различные исправления, внесённые после версии 10.21. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

Сообщество PostgreSQL прекратит выпуск обновлений серии 10.X в ноябре 2022 г. Поэтому пользователям не следует медлить с переходом на более новую основную версию.

E.28.1. Миграция на версию 10.22

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Если вы обновляете сервер с более ранней версии, чем 10.19, см. также Раздел E.31.

E.28.2. Изменения

- Предотвращение замены объекта при выполнении скрипта расширения, если этот объект до этого не принадлежал расширению (Том Лейн)

  Теперь скриптам расширения запрещается выполнять `CREATE OR REPLACE` для существующего объекта, не принадлежащего расширению. Также запрещается выполнять `CREATE IF NOT EXISTS` в аналогичной ситуации. Тем самым обеспечивается защита от атак с внедрением троянского кода, когда злонамеренный пользователь базы данных мог стать владельцем объекта в расширении, а затем изменить его с целью перехватить управление при его дальнейшем использовании другими пользователями. Попутно эта защита снижает риск случайной замены объектов.

  Проект PostgreSQL благодарит Свена Клемма за сообщение об этой проблеме. (CVE-2022-2625)

- Исправление воспроизведения записей WAL для `CREATE DATABASE` на ведомых серверах (Кётаро Хоригути, Асим Правин, Пол Гуо)

  На ведомых серверах могут отсутствовать каталоги табличных пространств, необходимые для воспроизведения записей WAL о создании базы данных. До этого исправления ведомый сервер не мог восстановиться в таких случаях, хотя каталоги могли отсутствовать на законных основаниях. Теперь табличное пространство создаётся (как обычный каталог), а по достижении согласованного состояния проверяется, что оно было корректно удалено.

- Поддержка «внутренних» табличных пространств (Томас Манро, Микаэль Пакье, Альваро Эррера)

  В PostgreSQL табличное пространство — это обычно символическая ссылка на каталог в другой файловой системе. Теперь табличное пространство может представлять собой простой каталог. Это не позволяет размещать таблицы в разных файловых системах, но полезно при тестировании. Кроме того, это является частью решения предыдущей проблемы, которое заключается в том, что отсутствующее табличное пространство временно создаётся в виде «внутреннего» пространства.

- Исправление проверки прав в `CREATE INDEX` (Натан Боссарт, Ной Миш)

  В результате исправления уязвимости CVE-2022-1552 команда `CREATE INDEX` стала искать классы операторов и другие объекты с правами владельца таблицы, а не с правами выполняющего её пользователя, как это было ранее. При этом возникала проблема с выгрузкой/восстановлением данных, поскольку pg_dump выполняет `CREATE INDEX` до назначения прав.

- Выполнение немедленной фиксации транзакции в протоколе расширенных запросов после `CREATE DATABASE` и других команд, которые нельзя использовать в блоке транзакции (Том Лейн)

Если клиент сразу после такой команды отправляет не сообщение Sync, а другую команду, любой сбой в последней приводит к откату предыдущей. Обычно следствием этого будет рассогласование состояния на диске (например, будет отсутствовать или обнаружится лишний каталог базы данных). Механизмы предотвращения такой несогласованности, как выяснилось, работают, когда несколько команд передаются в одном сообщении простого протокола, но не когда они передаются в последовательности сообщений расширенного протокола. Чтобы обеспечить согласованность и при этом не поломать сценарии использования, которые работали ранее, теперь после таких команд выполняется неявная фиксация транзакции.

- Устранение условий гонки при проверке видимости транзакций (Саймон Риггс)

  Функция `TransactionIdIsInProgress` могла выдать `false` для транзакции, которая ещё не стала видимой, что могло иметь различные негативные последствия. Как правило, окно условий гонки довольно узкое, но синхронная репликация расширяет его, поскольку в этом окне выполняется ожидание синхронной реплики.

- Устранение ошибки планировщика «variable not found in subplan target list» (переменная не найдена в целевом списке подплана) при подтягивании наверх вложенного `SELECT`, задействованного в аргументе функции `GROUPING` (Ричард Гуо)

- Исключение возможности получения от функции `pg_stat_get_subscription()` дополнительной строки с «мусорными» значениями (Кунтал Гхош)

- Реализация очистки состояния сеанса в `pg_stop_backup()` (Фудзии Масао)

  Раньше такая очистка не выполнялась, что могло приводить к ошибкам или сбоям проверочных утверждений в том же сеансе.

- Исправление сопоставления псевдонимов соединяемых отношений в предложениях `FOR [KEY] UPDATE/SHARE` (Дин Рашид)

  Ранее в особых случаях могло выдаваться неверное сообщение об ошибке.

- Устранение сбоя при использовании в конструкции `XMLTABLE` или `JSON_TABLE` слишком большого числа псевдонимов столбцов (Альваро Эррера)

- Предотвращение выполнения выражений и функций `ROW()` со слишком большим количеством столбцов во `FROM` (Том Лейн)

  Запросы с более чем 1600 столбцами не поддерживаются и всегда завершаются ошибкой. Однако обнаружилось, что старую реализацию можно было довести до сбоя проверочных утверждений или краха, составив запрос с более чем 32K столбцами. Чтобы предотвратить это, добавлена проверка количества столбцов при разборе запроса.

- Отображение столбцов `SELECT`, выводимых в выдаваемом определении правила или представления, с предложением `AS "?column?"`, если могут существовать внешние ссылки на них (Том Лейн)

  Ранее такой автоматически создаваемый псевдоним всегда скрывался, но вследствие этого в некоторых случаях полученное определение правила или представления нельзя было восстановить.

- Исправление выгрузки представления, вызывающего во `FROM` функцию с результатом составного типа, какие-либо столбцы которого были удалены после создания представления (Том Лейн)

  Ранее выгруженное представление могло содержать лишние псевдонимы для столбцов результата функции, что препятствовало восстановлению данных или работе pg_upgrade.

- Передача информации о неявном создании семейств операторов событийным триггерам (Масахико Савада)

Если при выполнении `CREATE OPERATOR CLASS` неявно создавалось семейство операторов, событийные триггеры не замечали создание этого объекта.

- Корректировка изменения управляющего файла при выполнении процедуры точки перезапуска во время повышения ведомого сервера (Кётаро Хоригути)

  Ранее при завершении процедуры точки перезапуска она могла ошибочно изменить сведения о последней контрольной точке в управляющем файле, следствием чего могла быть аварийная остановка сервера и невозможность перезапустить сервер в случае сбоя до выполнения следующей контрольной точки.

- Предотвращение срабатывания тайм-аута `wal_receiver_timeout` на ведомом сервере в процессе логической репликации больших транзакций (Вэй Ван, Амит Капила)

  Если во время большой транзакции на ведущем сервере ведомому не передавались никакие данные (например, потому что изменяемые в ней таблицы не опубликованы), на ведомом сервере мог сработать тайм-аут. Для устранения этой проблемы в таких ситуациях периодически будут передаваться сообщения об активности.

- Запрет вложенных операций резервного копирования в режиме передачи данных логической репликации (Фудзии Масао)

- Устранение утечки памяти в коде подписчика логической репликации (Хоу Чжицзе)

- Исключение временных таблиц, создаваемых при перезаписи кучи для материализованных представлений, из логической репликации (Эйлер Тавейра)

  Без каких-либо ограничений публикация `FOR ALL TABLES` имеет свойство охватывать и временные таблицы. Для их исключения применялась эвристика, но в ней забыли учесть внутренние временные таблицы, создаваемые при перезаписи материализованных представлений. Из-за этого могли возникать ошибки типа «logical replication target relation ... does not exist» (целевое отношение логической репликации ... не существует) во время выполнения `REFRESH MATERIALIZED VIEW`.

- Предотвращение утечки открытого дескриптора при чтении неправильного файла с сокращёнными обозначениями часовых поясов (Кётаро Хоригути)

  В таких случаях могли выдаваться некритичные предупреждения.

- Обеспечение поддержки дополнительных серверных параметров, для которых в качестве короткого описания задано NULL (Стив Чавес)

  Расширения могли создавать такие параметры и ранее, но в некоторых местах кода при их обработке происходил сбой.

- Исправление обработки флагов `BRIN_EVACUATE_PAGE` в логике проверки целостности WAL (Хайян Ван)

- Удаление в libpq ненужной проверки владельца файла SSL-ключа (Том Лейн)

  В предыдущих корректирующих выпусках мы скопировали правила проверки разрешений для файла закрытого ключа SSL, принятые на стороне сервера, в libpq. Однако проверку владения этого файла копировать не следовало. Хотя она работала в обычных ситуациях, она мешала запустить клиентское приложение от имени root, и, возможно, в других случаях.

- Обеспечение вывода вменяемого сообщения при потере подключения к серверу в ecpg (Том Лейн)

  Вследствие неправильной обработки ошибки, сформированной libpq, например при потере подключения, вместо полезного сообщения могла выдаваться строка «(null)», а в старых версиях мог произойти сбой.

- Предотвращение аварийного сбоя в ecpglib при выполнении операций не в ожидаемом порядке (Том Лейн)

Если определённые операции, например `EXEC SQL PREPARE`, выполнялись до установления подключения к базе данных, мог произойти сбой (вместо ошибки, как можно было ожидать).

- Исключение лишних вызовов `newlocale()` в ecpglib (Ной Миш)

  Раньше объекты локали C создавались и освобождались для каждого запроса, теперь же объект локали выделяется один раз для каждого процесса при первом подключении. Это устраняет утечку памяти libc в AIX и может несколько повысить общую производительность.

- Добавление в psql перевода строки после прерывания команды `\watch` сигналом Control-C (Павел Стехуле)

  Теперь libedit (а возможно и libreadline) будет точно знать, в каком столбце находится курсор.

- Устранение найденных в `contrib/pg_stat_statements` проблем с очень большими файлами текстов запросов на 32-битных платформах (Том Лейн)
- Обеспечение в `contrib/postgres_fdw` передачи констант типов `regconfig` и `reg*` с правильной схемой (Том Лейн)
- Блокировка сигналов во время выделения динамической общей памяти в Linux (Томас Манро)

  Это изменение позволяет избежать проблем при прерывании сигналом функции `posix_fallocate()`.

- Исправление обработки неожиданных ошибок `EEXIST`, выдаваемых функцией `shm_open()` (Томас Манро)

  Это предотвращает возможный сбой в Solaris.

- Корректировка теста PL/Perl для обеспечения совместимости с Perl 5.36 (Дагфинн Ильмари Маннсакер)
- Недопущение ошибочного использования устаревшей библиотеки libldap_r в случае, когда при сборке PostgreSQL обнаруживается несколько инсталляций OpenLDAP (Том Лейн)

E.29. Выпуск 10.21

Дата выпуска: 2022-05-12

В этот выпуск вошли различные исправления, внесённые после версии 10.20. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

Сообщество PostgreSQL прекратит выпуск обновлений серии 10.X в ноябре 2022 г. Поэтому пользователям не следует медлить с переходом на более новую основную версию.

E.29.1. Миграция на версию 10.21

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Если вы обновляете сервер с более ранней версии, чем 10.19, см. также Раздел E.31.

E.29.2. Изменения

- Расширение множества операций, выполняемых в «песочнице» как операции с ограничениями по безопасности (Сергей Шиндерук, Ной Миш)

  В коде автоочистки, `CLUSTER`, `CREATE INDEX`, `REINDEX`, `REFRESH MATERIALIZED VIEW` и pg_amcheck защитный механизм, обеспечивающий безопасное выполнение операции, включался слишком поздно или не включался вовсе при некоторых условиях. Пользователь, имеющий права для создания невременных объектов в базе данных, мог создать такой объект, который позволил бы выполнить произвольный SQL-код с правами суперпользователя при следующей обработке этого объекта процессом автоочистки или одной из перечисленных команд, запущенной суперпользователем.

Проект PostgreSQL благодарит Александра Лахина за сообщение об этой проблеме. (CVE-2022-1552)

- Прекращение обработки заданных в запросе псевдонимов столбцов в переменных «вся строка», ссылающихся на простые таблицы (Том Лейн)

  Столбцы в кортежах, образованных переменными «вся строка» (например, `tbl.*` во всех контекстах, кроме верхнего уровня списка `SELECT`), теперь всегда получают имена от соответствующего составного типа, если он имеется. Ранее мы пытались отслеживать все псевдонимы столбцов, заданные в элементе `FROM`, на который ссылается переменная. Но это сомнительно с точки зрения семантики, так как содержимое переменной при этом фактически вообще не соответствует составному типу, который она в принципе должна представлять. Предыдущие попытки устранить эту несогласованность имели плохие последствия, вплоть до сохранения нечитаемых данных на диске, поэтому было решено отказаться от этой идеи.

  В случаях, когда требуется переименовать такие столбцы, это можно сделать, добавив дополнительный уровень вложенности `SELECT`, чтобы переменная «вся строка» ссылалась на результат внутреннего `SELECT`, а не на исходную таблицу. Тогда переменная типа `record` будет получена из неё, что даст желаемый результат.

- Исправление некорректного вывода для типов `timestamptz` и `timetz` в `table_to_xmlschema()` и родственных функциях (Ренан Соарес Лопес)

  Выводимая xml-схема для этих типов включала неправильные регулярные выражения.

- Предотвращение аварийного сбоя в анализаторе при обработке предложения `VALUES` с нулём столбцов (Том Лейн)

- Исправление ошибок планировщика при обработке конструкций `GROUPING()`, обращающихся к внешним уровням запроса (Ричард Гуо, Том Лейн)

- Исправление построения плана со сканированием только индекса, в случаях, когда индексы содержат и возвращаемые, и невозвращаемые столбцы (Том Лейн)

  Предыдущая реализация могла попытаться выдать в дополнение к возвращаемым столбцам невозвращаемые, что по большому счёту было безвредно, так как выдаваемые фиктивные значения фактически никуда не попадали. Однако после недавнего добавления проверки, отвергающей такой план, это стало неприемлемым.

- Ликвидация утечки памяти на протяжении запроса в узле IndexScan, производящем пересортировку (Александр Каленик)

- Обеспечение в `ALTER FUNCTION` поддержки изменения в одной команде и свойства параллельности функции, и списка назначаемых для неё переменных (Том Лейн)

  Ранее изменение свойства параллельности терялось, если в той же команде для функции менялось предложение `SET`.

- Исправление ошибочной сортировки строк таблицы при выполнении `CLUSTER` по индексу, ведущим ключом которого является выражение (Питер Гейган, Томас Манро)

  Таблица перестраивалась с корректными данными, но порядок сортировки никак не соответствовал порядку индекса.

- Устранение условий гонки между `DROP TABLESPACE` и процессом контрольных точек (Натан Боссарт)

  Процедура контрольной точки, выполняемая принудительно командой `DROP TABLESPACE`, иногда могла не удалить все ставшие ненужными файлы из каталога табличного пространства, вследствие чего выдавалась ошибка «tablespace is not empty» (табличное пространство не пусто).

- Исключение возможной проблемы при восстановлении после сбоя, которому предшествовало выполнение команды `TRUNCATE`, наложившееся на контрольную точку (Кётаро Хоригути, Хейкки Линнакангас, Роберт Хаас)

  Процедура `TRUNCATE` должна задерживать завершение контрольной точки, пока файл таблицы на диске не будет обрезан. В противном случае при воспроизведении журнала с этой контрольной точки в предположительно освобождённых страницах могут обнаружиться лишние данные, что может вызвать ошибку воспроизведения.

- Исправление небезопасного обращения к данным TOAST во время очистки временных объектов (Андрес Фройнд)

  Процедура удаления временных объектов при завершении серверного процесса могла прерываться ошибкой «FATAL: cannot fetch toast data without an active snapshot» (ВАЖНО: прочитать данные TOAST без активного снимка нельзя). Обычно это не было проблемой, так как при следующем использовании этой временной схемы она должна была очиститься успешно.

- Устранение сбоя с сообщением «PANIC: xlog flush request is not satisfied» (ПАНИКА: запрос на сброс xlog не выполнен) при повышении ведомого сервера после того, как на нём была пропущена отсутствующая запись продолжения WAL (Сами Имсейх)

- Устранение возможности самоблокировки при обработке конфликта на сервере горячего резерва (Андрес Фройнд)

  При неудачном наложении событий процесс применения WAL мог зависнуть, ожидая, что какой-то другой процесс снимет блокировку буфера.

- Обеспечение возможности перезапуска рабочих процессов применения логической репликации даже при достижении предела `max_sync_workers_per_subscription` (Амит Капила)

  Вследствие ошибки в коде проверки этого предела перезапущенный рабочий процесс сразу же завершался, и в результате рабочих процессов оказывалось меньше, чем их должно было быть.

- Внесение в WAL значений столбцов, образующих ключ идентификации реплики, которые не были изменены при изменении строки, но хранятся вне кортежа (Дилип Кумар, Амит Капила)

  Если значения таких столбцов не записывать в WAL, подписчики не смогут получить их и не смогут реплицировать изменение.

- Улучшение сообщения об ошибке на стороне подписчика логической репликации при обращении к отношению неподдерживаемого типа (Том Лейн)

  Серверы версии 13 и новее поддерживают секционированные таблицы, тогда как серверы старых версий подписаться на такие таблицы не позволяют. При попытке сделать это ранее выдавалось вводящее в заблуждение сообщение «table XYZ not found on publisher» (таблица XYZ не найдена на сервере публикации). Теперь сообщение будет более осмысленным.

- Недопущение выполнения функций SPI во время компиляции функции на PL/Perl (Том Лейн)

  Perl позволяет выполнять определённый пользователем код во время компиляции функции на PL/Perl. Однако такому коду нельзя позволять выполнять операции SQL через SPI. В противном случае либо произойдёт сбой, либо возникнет угроза безопасности, так как мы на самом деле не ожидаем, что код будет выполняться во время проверки функции. Теперь во избежание этого добавлена проверка с понятным сообщением об ошибке.

- Обеспечение в libpq возможности принимать файлы закрытых ключей SSL, принадлежащие root (Дэвид Стил)

  Это изменение уравнивает действующие в libpq правила безопасного владения и разрешений для файлов ключей SSL с правилами, принятыми на стороне сервера, начиная с версии 9.6. А именно, к текущим правилам добавлен случай, когда файлом ключа владеет root и для этого

файла заданы разрешения `rw-r-----` или более строгие. Это облегчает управление файлами ключей на уровне системы.

- Добавление в pg_ctl перепроверки существования процесса postmaster во время ожидания действий stop/restart/promote (Том Лейн)

  Ранее утилита pg_ctl проверяла, работает ли postmaster, попутно с тем, как передавала ему сигнал stop или promote, но затем она просто наивно ждала, пока не изменится состояние на диске. Если процесс postmaster завершался нештатно, не удалив свой файл PID или не отразив состояние в управляющем файле, ожидание pg_ctl завершалось только по тайм-ауту. Теперь pg_ctl будет постоянно перепроверять, не исчез ли процесс postmaster.

- Обеспечение в функциях модуля `contrib/pageinspect` корректной обработки полностью нулевых страниц (Микаэль Пакье)

  Такие страницы нетипичны, но вполне легитимны, однако этот модуль не умел их обрабатывать. Теперь функции модуля, получив такую страницу, возвращают NULL или пустой набор строк, что видится более полезным поведением, чем выдавать ошибку.

- Добавление в `contrib/pageinspect` защиты от некорректного содержимого «специальной области» страниц, усиление проверок размера страниц и добавление недостающих проверок соответствия типа индекса ожидаемому (Микаэль Пакье, Джастин Призби, Жюльен Руо)

  После этих изменений вызвать падение модуля, передав ему плохие данные, будет не так просто.

- Добавление в `contrib/postgres_fdw` проверки возможности безопасного удалённого выполнения предложения `ORDER BY` до передачи запроса с удалённой сортировкой, а также дополнение этого предложения указанием `USING`, когда оно необходимо (Ронан Данклау)

  До этого удалённый сервер мог выполнять сортировку не в том порядке, в котором ожидалось. Хотя в каких-то случаях это не имело большого значения, в других могли выдаваться совершенно неверные результаты, если удалённые данные поступали на вход локальной процедуре соединения слиянием.

- Устранение разнообразных дефектов, выявленных проверками clang `-fsanitize=undefined` (Том Лейн, Андрес Фройнд, Чжихун Юй)

  Большинство дефектов исправлено только ради формального соответствия букве языка C и стандартов POSIX, вряд ли они могли как-то проявляться в производственных сборках.

- Внесение исправлений в код PL/Perl, чтобы он собирался компиляторами C, не поддерживающими вложение операторов в выражения (Том Лейн)

- Устранение потенциальной ошибки сборки pg_dumpall в Windows при использовании для сборки не MSVC (Андрес Фройнд)

- Смена применяемой в сборках для Windows программы, создающей файлы DEF, с pexports на gendef (Эндрю Дунстан)

  Таким образом обеспечена совместимость процесса сборки с последней версией инструментария MSys.

- Предотвращение дополнительного разворачивания шаблонов оболочки в программах, собираемых в MinGW (Эндрю Дунстан)

  По какой-то причине библиотека C, представленная в MinGW, по умолчанию разворачивает символы подстановки, полученные в командной строке программы. Такое поведение вызывает вопросы, как минимум потому что с библиотекой MSVC этого не происходит, поэтому оно отключено.

- Обновление данных часовых поясов до версии tzdata 2022a, включающее изменение правил перехода на летнее время в Палестине, а также корректировку исторических данных для Чили и Украины.

E.30. Выпуск 10.20

Дата выпуска: 2022-02-10

В этот выпуск вошли различные исправления, внесённые после версии 10.19. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

Сообщество PostgreSQL прекратит выпуск обновлений серии 10.X в ноябре 2022 г. Поэтому пользователям не следует медлить с переходом на более новую основную версию.

E.30.1. Миграция на версию 10.20

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Если вы обновляете сервер с более ранней версии, чем 10.19, см. также Раздел E.31.

E.30.2. Изменения

- Исправление планов сканирования только индекса, работавших некорректно, когда могут быть извлечены не все столбцы индекса (Том Лейн)

  В случае, когда индекс содержит и извлекаемые, и неизвлекаемые столбцы, и один из неизвлекаемых столбцов — выражение, использующее столбец из числа извлекаемых из индекса, для запроса, использующего этот индекс, мог быть построен план со сканированием только индекса, рассчитывающий прочитать неизвлекаемый столбец, вместо того, чтобы вычислить выражение, используя значение извлекаемого столбца. Вследствие этой ошибки неизвлекаемый столбец получал значение NULL, и результаты запроса могли оказаться некорректными.

- Представление операции приведения к типу с неопределённым идентификатором в виде узла RelabelType вместо вызова функции с приведением длины (Том Лейн)

  Хотя и функция приведения должна сделать всё правильно (не делать ничего), её вызов неэффективен и потому нежелателен.

- Исправление сбоя при воспроизведении WAL в случае, когда состояние согласованности достигается точно на границе страницы WAL (Альваро Эррера)

- Исправление запуска физической репликации в ситуации, когда произошло зацикливание идентификаторов транзакций (Абхиджит Менон-Сен, Томаш Вондра)

  При запуске реплицирующего сервера в момент, когда последовательность активных транзакций на ведущем пересекает границу зацикливания транзакций (так что у некоторых последних транзакций значения XID оказываются меньше, чем у предшествующих), процесс репликации прерывался с ошибкой «out-of-order XID insertion in KnownAssignedXids» (нарушение порядка при добавлении XID в KnownAssignedXids). Затем процесс репликации перезапускался, но не мог продвинуться дальше этой ошибки.

- Устранение лексических ограничений для SQL-команд, выполняемых через подключение логической репликации (Том Лейн)

  Процесс walsender выдавал ошибку при передаче SQL-команд: содержащих точку с запятой без кавычек; содержащих константы в долларах с нечётным количеством апострофов или кавычек; начинающихся с комментария. Более того, в результате дефектной обработки ошибок и последующие команды могли не выполняться из-за неожиданных ошибок.

- Предотвращение потери времени фиксирования последней подтранзакции для транзакции (Алекс Кингсборо, Кётаро Хоригути)

- Обеспечение синхронизации с ФС (`fsync`) подкаталога `pg_logical/mappings` во время контрольных точек (Натан Боссарт)

В некоторых файловых системах из-за отсутствия синхронизации файлы состояния логической перезаписи могли потеряться при сбое системы.

- Построение расширенной статистики для секционированных таблиц (Джастин Призби)

  С предшествующим исправлением было намеренно отключено построение статистики для деревьев наследования в старом стиле, но при этом также были затронуты секционированные таблицы, хотя для них такое ограничение необоснованно. В результате данного исправления операция `ANALYZE` сможет вычислять значения для объектов статистики, относящихся к секционированным таблицам. (Но учтите, что при автоочистке секционированные таблицы таким образом не обрабатываются, поэтому вы должны периодически производить `ANALYZE` с секционированной таблицей вручную, если вам нужна эта статистика в актуальном состоянии.)

- Исключение расширенной статистики из рассмотрения при обработке деревьев наследования (Джастин Призби)

  В настоящее время значения расширенной статистики вычисляются только для отдельных таблиц, а не для деревьев наследования целиком. Однако эти значения ошибочно учитывались при планировании запросов, обращающихся к деревьям наследования, в результате чего могли получаться оценки хуже, чем оценки по умолчанию.

- Недопущение изменения типа столбцов секционированной таблицы в случае, когда тип кортежа таблицы используется где-то ещё как составной тип (Том Лейн)

  Соответствующее ограничение давно существовало для обычных таблиц, но для секционированных таблиц нужная проверка отсутствовала.

- Недопущение операции `ALTER TABLE ... DROP NOT NULL` со столбцом, включённым в индекс, который используется для идентификации реплики (Хайин Тан, Хоу Чжицзе)

  Тот же запрет существовал ранее для индексов первичных ключей.

- Обеспечение корректного обновления кешируемого состояния таблицы при переключении индекса, используемого для идентификации реплики (Хайин Тан, Хоу Чжицзе)

  В одновременно выполняющихся сеансах могли оказываться разные представления о том, какой индекс идентифицирует реплику, что могло нарушить работу логической репликации.

- Устранение утечек памяти при выполнении операций `REASSIGN OWNED BY`, меняющих владельца для множества объектов (Джастин Призби)

- Исправление представления переменных, указывающих на всю строку, в правилах `INSERT ... VALUES` (Том Лейн)

  Переменная, указывающая на строку, представлялась в виде «var.*», но при перезагрузке правила эта запись могла развернуться в список отдельных столбцов, что меняло её семантику. Во избежание этого запись переменной дополнена явным приведением, как это сделано в остальных местах.

- Исправление и удаление ряда некорректных проверочных утверждений (Саймон Риггс, Микаэль Пакье, Александр Лахин)

  Устранённые недочёты проявлялись только в отладочных сборках.

- Устранение условий гонки, в которых не получалось локализовать сообщения об ошибках, выдаваемые на ранних этапах использования libpq или ecpglib в многопоточном режиме (Том Лейн)

- Ликвидация вызова `strerror` в функции libpq `PQcancel` (Том Лейн)

  Функция `PQcancel` должна быть безопасной для вызова из обработчика сигнала, но использованная в ней функция `strerror` безопасной не является. Такое небезопасное

использование могло иметь последствия только в маловероятном случае ошибки при передаче сообщения отмены серверу, что может объяснить отсутствие жалоб на эту проблему со стороны пользователей.

- Выбор для команды psql `\password` в качестве целевого пользователя по умолчанию пользователя `CURRENT_USER`, а не пользователя, изначально установившего подключение (Том Лейн)

  Это соответствует документированному поведению и позволяет исключить проблему отсутствия прав в случае выполнения `SET ROLE` или `SET SESSION AUTHORIZATION` после установления подключения. Во избежание путаницы теперь в приглашении для смены пароля выводится имя целевого пользователя.

- Недопущение вызова `gettext()` в обработчиках сигнала Control-C в psql и некоторых других клиентских программах (Том Лейн)

  Хотя с мест не поступали сообщения о связанных с этим проблемах, очень маловероятно, что такие вызовы безопасны.

- Предоставление возможности отменить начальный запрос пароля в pg_receivewal и pg_recvlogical (Том Лейн, Натан Боссарт)

  Ранее эти программы нельзя было прервать сигналом Control-C, когда они запрашивали пароль.

- Исправление в pg_dump порядка выгрузки определённых пользователем приведений (Том Лейн)

  В редких случаях в выводимом скрипте команда, задействующая пользовательское приведение, могла оказаться перед командой, его создающей.

- Исправление диагностики и обработки ошибок в pg_dump и pg_basebackup (Том Лейн)

  Прежняя реализация не контролировала ошибки после вызовов ядра в одних случаях и могла выдавать неправильные значения errno в других.

- Корректировка результатов сканирования только индекса при использовании индексов `contrib/btree_gist`, построенных по столбцам `char(`*`N`*`)` (Том Лейн)

  Ранее при сканировании только индекса значения столбцов выдавались без конечных пробелов, что не является ожидаемым поведением, но именно так данные хранились в этих индексах. В результате исправления код будет сохранять значения `char(`*`N`*`)` с нужным количеством дополняющих пробелов. Поведение такого индекса не изменится сразу само, если вы не переиндексируете его, однако и без переиндексации значения без конечных пробелов постепенно заменятся полными в процессе изменения данных. На запросы, в которых не использовались планы со сканированием только индекса, это не влияет в любом случае.

- Смена используемого в configure модуля Python, помогающего определить параметры сборки PL/Python, с устаревшего distutils на sysconfig (Питер Эйзентраут, Том Лейн, Андрес Фройнд)

  Благодаря этому, теперь с Python 3.10 при выполнении configure не выдаются предупреждения о том, что модуль distutils устарел и его планируется удалить в Python 3.12. С выходом 3.12 прежняя реализация configure --with-python предположительно просто не будет работать. Эта подготовка к будущему имеет и обратную сторону: модуль sysconfig появился только в Python 2.7 и в Python 3.2 (в линейке Python 3), поэтому теперь собрать PL/Python с очень старыми версиями Python будет невозможно.

- Устранение проблемы компиляции PL/Perl в Windows при использовании Perl 5.28 и новее (Виктор Вагнер)

- Устранение проблемы компиляции PL/Python при использовании Python 3.11 и новее (Питер Эйзентраут)

- Обеспечение поддержки компиляции с использованием Visual Studio 2022 (Ханс Бушман)
- Реализация возможности вызывать скрипты-обёртки `.bat` в нашей сборочной среде MSVC, не переходя в содержащий их каталог (Антон Волошин, Эндрю Дунстан)

E.31. Выпуск 10.19

Дата выпуска: 2021-11-11

В этот выпуск вошли различные исправления, внесённые после версии 10.18. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.31.1. Миграция на версию 10.19

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Однако учтите, что в инсталляциях, где используется физическая репликация, сначала надо обновлять резервные серверы, а затем главный, как описано ниже в третьем пункте списка изменений.

Кроме того, было обнаружено несколько ошибок, которые приводили к повреждениям индексов, что освещено в следующих нескольких пунктах списка изменений. Если какие-либо из описанных проблем могут касаться вас, после обновления рекомендуется перестроить возможно повреждённые индексы.

Если вы обновляете сервер с более ранней версии, чем 10.16, см. также Раздел E.34.

E.31.2. Изменения

- Недопущение обработки сервером посторонних данных после сообщений установления шифрования SSL или GSS (Том Лейн)

  Злоумышленник, имеющий возможность внедрять данные в TCP-соединение, мог вставить некоторый объём незашифрованных данных в начале сеанса, который считался зашифрованным. Эксплуатируя это, можно было передавать серверу подставные команды, хотя это могло сработать, только если сервер не запрашивал никакие данные для аутентификации. (Однако опасность существовала и в случае, когда сервер осуществлял аутентификацию по сертификату.)

  Проект PostgreSQL благодарит Джейкоба Чемпиона за сообщение об этой проблеме. (CVE-2021-23214)

- Недопущение обработки библиотекой libpq посторонних данных после сообщений установления шифрования SSL или GSS (Том Лейн)

  Злоумышленник, имеющий возможность внедрять данные в TCP-соединение, мог вставить некоторый объём незашифрованных данных в начале сеанса, который считался зашифрованным. Эксплуатируя это, можно было передать поддельные ответы на несколько первых запросов сервера, хотя ввиду других особенностей поведения libpq реализовать это на практике было не так просто. Была возможна и другая линия атаки — перехват пароля клиента или других секретных данных, которые могли передаваться в сеансе ранее. Возможность таких атак была продемонстрирована с сервером, подверженным уязвимости CVE-2021-23214.

  Проект PostgreSQL благодарит Джейкоба Чемпиона за сообщение об этой проблеме. (CVE-2021-23222)

- Исправление работы физической репликации в случаях нештатной остановки главного сервера после передачи сегмента WAL с неполной записью WAL (Альваро Эррера)

  Если главный сервер отключался, не успев завершить сохранение окончания неполной записи WAL, прежний алгоритм восстановления после сбоя приводил к тому, что WAL

перезаписывался с начала неполной записи. Это представляло проблему, так как резервные серверы могли уже получить копии этого сегмента WAL. Затем они получали не согласующийся с ним следующий сегмент, и не могли продолжить работу без ручного вмешательства. Для решения этой проблемы теперь в ходе восстановления при перезапуске не будет допускаться пересечение границы сегмента WAL. Вместо этого в начало следующего сегмента WAL будет записываться запись нового типа, сообщающая получателям о том, что неполная запись WAL не будет завершена никогда и должна быть отброшена.

Установку данного обновления рекомендуется начинать с резервных серверов, чтобы в случае сбоя главного они были готовы обработать новый тип записи WAL, а затем обновлять главный.

- Исправление в `CREATE INDEX CONCURRENTLY` ожидания последних подготовленных транзакций (Андрей Бородин)

  Строки, добавленные только что подготовленными транзакциями, могли не попасть в новый индекс, вследствие чего запросы, полагающиеся на этот индекс, могли пропускать их. Предыдущее исправление проблем такого рода не учитывало, что команды `PREPARE TRANSACTION` могли ещё выполняться в тот момент, когда команда `CREATE INDEX CONCURRENTLY` проверяла их. Как и ранее, в инсталляциях, где включены подготовленные транзакции (`max_prepared_transactions` > 0), рекомендуется переиндексировать все индексы, построенные неблокирующим способом, на случай, если при их построении возникла подобная ситуация.

- Исключение условий гонки, в которых обслуживающие процессы могли не добавить записи для новых строк в индекс, создаваемый неблокирующим способом (Ной Миш, Андрей Бородин)

  Хотя на практике такие условия представляются маловероятными, эта проблема могла затронуть любой индекс, создаваемый или перестраиваемый с указанием `CONCURRENTLY`. Все такие индексы рекомендуется перестроить, чтобы они в любом случае стали корректными.

- Исправление хеш-функций для типов `float4` и `float8`, чтобы они возвращали одинаковые результаты для значений NaN (Том Лейн)

  Так как в PostgreSQL значения NaN разных типов с плавающей точкой считаются равными, важно, чтобы и хеш-функции выдавали одинаковые хеши для всех комбинация битов, которые воспринимаются как NaN согласно стандарту IEEE 754. Раньше этого не происходило, вследствие чего хеш-индексы и планы запросов, использующие хеши, могли выдавать неправильные результаты для неканонических представлений NaN. (Подобное значение на большинстве машин можно было получить, записав `'-NaN'::float8`.) Поэтому сейчас рекомендуется переиндексировать хеш-индексы, построенные по столбцам с числами с плавающей точкой, если в них могли оказаться такие значения.

- Предотвращение потери данных в процессе восстановления после сбоя, произошедшего после выполнения `CREATE TABLESPACE`, при значении `wal_level` = `minimal` (Ной Миш)

  Если в промежутке между выполнением `CREATE TABLESPACE` и следующей контрольной точкой происходил сбой сервера, в ходе восстановления полностью удалялось содержимое каталога нового табличного пространства в расчёте на то, что при последующем воспроизведении WAL будет восстановлено всё в этом каталоге. Однако это плохо работает с оптимизациями, которые пропускают запись WAL (например, когда команда `COPY` записывает данные в создаваемую ей же таблицу). Такие оптимизации применяются, только если для `wal_level` установлено значение `minimal` (которое не является значением по умолчанию, начиная с 10 версии).

- Обеспечение аннулирования кеша отношений для таблицы, присоединяемой к секционированной таблице или отсоединяемой (Амит Ланготе, Альваро Эррера)

  В результате допущенного упущения последующие операции добавления/изменения, обращённые непосредственно к этой секции, могли выполняться некорректно, но только в рамках уже существующих сеансов.

- Обеспечение аннулирования кеша отношений для всех секций секционированной таблицы, добавляемой или удаляемой из публикации `FOR ALL TABLES` (Хоу Чжицзе, Вигнеш Си)

  В результате упущения репликация могла работать некорректно до завершения всех сеансов, существовавших на момент операции.

- Сохранение операции приведения к тому же типу в случае отсутствия модификатора типа (Том Лейн)

  Например, для столбца `f1` типа `numeric(18,3)` анализатор запроса просто отбрасывал приведение вида `f1::numeric`, полагая, что оно никак не влияет на выполнение запроса. Это так, но итоговым типом выражения всё же должен считаться простой `numeric`, а не `numeric(18,3)`. Это важно для правильного разрешения типов во внешних конструкциях, например в рекурсивных `UNION`.

- Запрещение создания правил сортировки, использующих ICU, когда это не поддерживается кодировкой базы (Том Лейн)

  Ранее такое правило можно было создать, но его нельзя было использовать из-за особенностей поиска правил сортировки; мало того, его нельзя было и удалить.

- Устранение потери точности в функции `power()` при передаче ей особых значений (Дин Рашид)

  Результат этой функции мог быть неточным, когда первый аргумент был примерно равен 1.

- Исключение ошибок в регулярных выражениях с захватом скобок внутри `{0}` (Том Лейн)

  При обработке выражений вида `(.){0}...\1` выдавалась ошибка «invalid backreference number» (неправильный номер обратной ссылки). Однако другие процессоры регулярных выражений, например Perl, воспринимают это, так же как некоторые подобные выражения воспринимает и наша реализация. Что хуже, вместо ошибки мог произойти сбой проверочного утверждения. После этого исправления такая ошибка выдаваться не будет, а обратная ссылка будет просто считаться не соответствующей ничему.

- Предотвращение некорректного установления соответствий для обратных ссылок в регулярных выражениях при некоторых условиях (Том Лейн)

  Ранее процессор регулярных выражений не заботился об очистке данных соответствия для захваченных скобок, отбрасывая частичное соответствие. В результате затем могло устанавливаться соответствие обратной ссылки там, где его не должно было быть из-за отсутствия определения ссылающегося элемента.

- Устранение ошибки, влияющей на производительность регулярных выражений с обратными ссылками внутри узлов итерации (Том Лейн)

  Из-за некорректной логики обработки обратных ссылок время поиска соответствия могло увеличиваться экспоненциально. К счастью, эта проблема в большинстве случаев сглаживалась другими оптимизациями.

- Исправление результатов выражения `AT TIME ZONE`, применённого к значению типа `time with time zone` (Том Лейн)

  Если целевой часовой пояс задавался динамическим сокращённым обозначением (то есть обозначением, равнозначным полному названию часового пояса, а не фиксированным смещением от UTC), результаты такого выражения были некорректными.

- Отказ от использования одной лишь статистики MCV для оценивания диапазона значений в столбце (Том Лейн)

  В некоторых особых случаях процедура `ANALYZE` строит список самых частых значений (MCV), но не строит гистограмму, хотя список MCV не даёт представления обо всех существующих значениях. В таких случаях планировщик не должен использовать только список MCV для оценивания диапазона значений в столбце.

- Обеспечение корректной очистки состояния при отмене транзакции после того, как был экспортирован её снимок (Дилип Кумар)

  Устранённая теперь ошибка могла создавать проблемы, только если тот же сеанс пытался ещё раз экспортировать снимок. Наиболее вероятный сценарий возникновения этой ошибки — создание слота репликации (за которым следует откат транзакции), а затем создание ещё одного слота репликации.

- Предотвращение зацикливания при отслеживании переполнения подтранзакций на ведомых серверах (Кётаро Хоригути, Александр Коротков)

  Следствием устранённого теперь дефекта могло быть значительное снижение производительности, которое проявлялось в излишнем трафике SubtransSLRU, на ведомых серверах.

- Обеспечение правильного учёта подготовленных транзакций во время повышения ведомого сервера (Микаэль Пакье, Андрес Фройнд)

  Ранее существовало небольшое окно, в котором подготовленные транзакции могли выпасть из снимка, создаваемого параллельно выполняющимся сеансом. Если данный снимок затем использовался в том сеансе для изменения данных, следствием ошибки могло быть повреждение данных или получение некорректных результатов.

- Исправление обнаружения ситуации, когда отношение достигает максимально допустимого размера (Том Лейн)

  Попытка расширить таблицу или индекс сверх предела в 2^32-1 блоков пресекалась и раньше, но с некоторым запозданием, так что успевала нарушиться согласованность внутреннего состояния.

- Корректировка отслеживания наличия CTE, изменяющих данные, при разворачивании правила `DO INSTEAD` (Грег Нанкарроу, Том Лейн)

  Из-за устранённого теперь упущения могли возникнуть проблемы, например, выбирались параллельные планы, когда это было небезопасно.

- Исправление формирования сообщения об ошибке, связанной с правами доступа к объектам расширенной статистики (Томаш Вондра)

  Ранее вместо нужного сообщения выдавалось «cache lookup error» (ошибка поиска в кеше).

- Исправление некорректной обработки снимков в параллельных исполнителях (Грег Нанкарроу)

  Из-за ошибки параллельные запросы могли работать неправильно на уровне изоляции транзакций ниже `REPEATABLE READ`.

- Обеспечение создания процессами walreceiver всех требующихся файлов уведомлений архива перед выходом (Фудзии Масао)

  Если приёмник WAL завершал свою работу ровно на границе сегмента WAL, он не создавал файл уведомления для последнего полученного сегмента, вследствие чего архивирование этого сегмента на ведомом сервере откладывалось.

- Исключение попыток блокирования псевдоотношений `OLD` и `NEW` в правиле, использующем `SELECT FOR UPDATE` (Масахико Савада, Том Лейн)

- Корректировка обработки агрегатных предложений `FILTER` в анализаторе запросов (Том Лейн)

  Если выражением `FILTER` являлась просто ссылка на логический столбец, семантический уровень агрегатной конструкции мог определяться неправильно, что приводило к поведению, не соответствующему стандарту. Если выражение `FILTER` само содержит агрегат, выдающий

логическое значение, должна выдаваться ошибка, но этого не происходило, что было чревато сбоем во время выполнения.

- Исключение обращений по нулевому указателю, приводящим к сбою, при удалении роли, владеющей объектами, которые удаляются параллельно (Альваро Эррера)
- Устранение предупреждения «snapshot reference leak» (утечка ссылки на снимок) при ошибке в `lo_export()` или связанной функции (Хейкки Линнакангас)
- Исправление подсчёта операций сканирования индексов SP-GiST в статистических представлениях (Том Лейн)

  Увеличение счётчика «количество сканирований индекса» в коде SP-GiST не производилось из-за упущения, хотя счётчики кортежей увеличивались корректно.
- Пересчёт соответствующих интервалов ожидания в случаях, когда параметр `recovery_min_apply_delay` меняется во время восстановления (Соумйадип Чакраборти, Ашвин Агравал)
- Устранение бесконечного цикла в случае добавления в хеш-таблицу в `simplehash.h` 2^32 элементов (Юрий Соколов)

  Вероятность столкнуться с этой ошибкой на практике невелика, так как для этого при существующем варианте использования `simplehash.h` потребуется задать в `work_mem` объём в сотни гигабайт.
- Уменьшение потребления памяти при вычислении расширенной статистики (Джастин Призби, Томаш Вондра)
- Исправление поведения ecpg при ошибке в `malloc()` в процессе установления соединения (Микаэль Пакье)
- Реализация выхода из самого внешнего блока подпрограммы на PL/pgSQL при выполнении `EXIT` (Том Лейн)

  Если в подпрограмме не требуется явный оператор `RETURN`, такое использование `EXIT` должно быть возможным, но ранее оно не допускалось.
- Устранение в pg_ctl жёстких ограничений на общую длину формируемых команд (Фил Крылов)

  Тем самым, например, устраняется ограничение на количество параметров командной строки, которое может быть передано программе postmaster. Отдельные пути, с которыми имеет дело pg_ctl, например путь к исполняемому файлу postmaster или путь к каталогу данных, в большинстве случае по-прежнему ограничены размером `MAXPGPATH` байт.
- Исправление в pg_dump выгрузки прав по умолчанию, определяемых не глобально (Нейл Чен, Масахико Савада)

  Если глобальная (неограниченная) команда `ALTER DEFAULT PRIVILEGES` отзывала какие-либо представляемые по умолчанию права, например право `EXECUTE` для функций, а затем ограниченная команда `ALTER DEFAULT PRIVILEGES` возвращала эти права отдельной роли или схеме, программа pg_dump не могла выгрузить ограниченное назначение прав правильно.
- Установление в pg_dump разделяемых блокировок для секционированных таблиц, подлежащих выгрузке (Том Лейн)

  Отсутствие блокировок обычно не имело негативных последствий, так как полученных программой pg_dump блокировок для конечных секций было достаточно, чтобы не допустить значимых DDL-операций с самой секционированной таблицей. Однако проблемы могли возникнуть при выгрузке секционированной таблицы, не имеющей потомков, так как в этом случае не устанавливались никакие блокировки.
- Увеличение производительности pg_dump за счёт исключения запросов политик RLS для каждой отдельной таблицы и устранения повторяющихся вызовов `format_type()` (Том Лейн)

После этих изменений при выгрузке данных с локального сервера производительность увеличивается лишь незначительно, тогда как при работе с удалённым сервером она может заметно возрасти вследствие минимизации сетевого взаимодействия.

- Исправление имени файла в сообщении об ошибке, выдаваемом pg_restore при получении неправильного файла с перечнем больших объектов (Даниэль Густафссон)
- Исправление ошибки в индексах `contrib/btree_gin` по столбцам `"char"` (не `char(`*`n`*`)`), проявлявшейся при сканировании индекса с использованием оператора < или <= (Том Лейн)

  Ранее при таком сканировании выдавались не все элементы, которые должны были.
- Реализация в расширении `contrib/pg_stat_statements` чтения его файла «текстов запросов» блоками размером не больше 1 ГБ (Том Лейн)

  Файлы очень большого размера с текстами запросов весьма нетипичны, но в случае их наличия ранее происходил сбой в 64-разрядной ОС Windows (в которой один запрос на чтение файла не может прочитать больше 2 ГБ).
- Устранение в `contrib/postgres_fdw` обращения по нулевому указателю при попытке сообщить об ошибке преобразования данных (Том Лейн)
- Добавление поддержки циклических блокировок для архитектуры RISC-V (Марек Шуба)

  Это имеет большое значение для обеспечение хорошей производительности на этой платформе.
- Обеспечение поддержки OpenSSL 3.0.0 (Питер Эйзентраут, Даниэль Густафссон, Микаэль Пакье)
- Установка правильного идентификатора типа для объектов BIO (абстракции ввода/вывода), которые создаёт PostgreSQL (Итамар Гафни)

  Исправленная теперь ошибка могла быть заметна только для кода, который выполняет такие задачи, как аудит инсталляции OpenSSL. Но всё же формально это было нарушением OpenSSL API, которое стоило устранить.
- Исправление в `pg_regexec()` обработки параметра *`search_start`*, выходящего за допустимые пределы (Том Лейн)

  Когда позиция *`search_start`* выходит за конец строки, теперь выдаётся результат `REG_NOMATCH`, тогда как раньше мог произойти сбой. Достижение этого состояния при использовании только ядра PostgreSQL не представляется возможным, однако расширения могут недостаточно строго проверять значение этого параметра.
- Обеспечение возможности вызова `GetSharedSecurityLabel()` в только что начатом сеансе, в котором ещё не получены критически важные для него элементы кеша отношений (Джефф Дэвис)
- Использование данных проекта CLDR для сопоставления принятых в Windows названий часовых поясов с часовыми поясами IANA (Том Лейн)

  Программа initdb, запущенная в Windows, пытается установить для нового кластера в параметре `timezone` часовой пояс IANA, соответствующий часовому поясу, выбранному в системе. Мы использовали таблицу соответствия, которая была сформирована несколько лет назад и обновлялась бессистемно: неудивительно, что в ней содержался ряд неточностей и не хватало добавленных не так давно часовых поясов. Выяснилось, что в CLDR отслеживаются наиболее подходящие сопоставления, поэтому решено использовать их данные. Это изменение затронет только впоследствии создаваемые кластеры, на существующие оно никак не повлияет.
- Обновление данных часовых поясов до версии tzdata 2021e, включающее изменение правил перехода на летнее время в Иордании, Палестине, на Фиджи и Самоа, а также корректировку исторических данных для Барбадоса, островов Кука, Ниуэ, Гайаны, Португалии и Тонга.

Кроме того, пояс Pacific/Enderbury был переименован в Pacific/Kanton. Помимо этого, следующие пояса были включены в соседние, более популярные пояса, в которых время было таким же с 1970 г.: Africa/Accra, America/Atikokan, America/Blanc-Sablon, America/Creston, America/Curacao, America/Nassau, America/Port_of_Spain, Antarctica/DumontDUrville и Antarctica/Syowa. Для всех этих поясов старое название сохранено в качестве альтернативного.

E.32. Выпуск 10.18

Дата выпуска: 2021-08-12

В этот выпуск вошли различные исправления, внесённые после версии 10.17. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.32.1. Миграция на версию 10.18

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Если вы обновляете сервер с более ранней версии, чем 10.16, см. также Раздел E.34.

E.32.2. Изменения

- Полное отключение повторного согласования SSL (Микаэль Пакье)

  Собственно повторное согласование SSL было отключено довольно давно, но сервер всё же мог продолжать взаимодействие с клиентом, получив от него запрос повторного согласования. При этом злонамеренно сконструированный запрос повторного согласования мог вызвать крах сервера (см. описание проблемы OpenSSL CVE-2021-3449). Теперь эта функциональность полностью отключена во всех версиях OpenSSL, где это возможно, а именно, в версии 1.1.0h и новее.

- Запрещение конструкции `SELECT ... GROUP BY GROUPING SETS (()) FOR UPDATE` (Том Лейн)

  Эта конструкция не должна допускаться, так же как не допускается `FOR UPDATE` с простым `GROUP BY`, но в соответствующей проверке пустые наборы группирования не обрабатывались корректно. В конечном итоге это могло привести к обращению по нулевому указателю в коде исполнителя.

- Недопущение случаев, когда запрос в `WITH` в результате переписывания сводится к просто `NOTIFY` (Том Лейн)

  Ранее в таких случаях происходило падение сервера.

- Округление результата умножения `numeric`, когда в нём оказывается больше 16383 знаков после точки, что раньше вызывало переполнение (Дин Рашид)

- Устранение ошибок и потери точности в особых случаях при возведении значений `numeric` в очень большую степень (Дин Рашид)

- Ликвидация ошибки деления на ноль в функции `to_char()` с форматом `EEEE` и входным значением `numeric` меньше 10^(-1001) (Дин Рашид)

- Реализация в `pg_size_pretty(bigint)` округления отрицательных значений в соответствии с округлением положительных (и с округлением, производимым версией с `numeric`) (Дин Рашид, Дэвид Роули)

- Устранение ошибки при вызове `pg_filenode_relation(0, 0)` — теперь при таком вызове выдаётся NULL (Джастин Призби)

- Блокирование расширения в `ALTER EXTENSION` при удалении или добавлении объекта, относящегося к этому расширению (Том Лейн)

Предыдущая реализация позволяла выполнить ALTER EXTENSION ADD/DROP одновременно с DROP EXTENSION, что могло вызвать сбой или повреждение записей в каталоге.

- Недопущение пустого имени слота в ALTER SUBSCRIPTION (Ли Япинь)
- Предупреждение конфликтов псевдонимов в запросах, генерируемых командами REFRESH MATERIALIZED VIEW CONCURRENTLY (Том Лейн, Бхарат Рупиредди)

  Ранее эта команда выдавала ошибку с материализованными представлениями, содержащими столбцы с определёнными именами, в частности, mv и newdata.
- Исправление в PREPARE TRANSACTION проверки конфликтующих блокировок, существующих в рамках сеанса и в рамках транзакции (Том Лейн)

  Транзакция не может быть подготовленной, если в ней существуют рекомендательные блокировки одного ID и на уровне сеанса, и на уровне транзакции. Ранее это ограничение не проверялось в полной мере, вследствие чего во время PREPARE TRANSACTION могло возникнуть состояние ПАНИКА.
- Исправление некорректного поведения DROP OWNED BY в случае множественного указания целевой роли в политике RLS (Том Лейн)
- Устранение ненужных проверок корректности при удалении роли из политики RLS командой DROP OWNED BY (Том Лейн)

  В частности, ранее в некоторых случаях безосновательно требовалось, чтобы DROP OWNED BY выполнял только суперпользователь.
- Транзакционное изменение флагов состояния индекса (Микаэль Пакье, Андрей Лепихов)

  Данное исправление устраняет ошибки при вычислении в индексах таких предикатов, которые фактически не являются постоянными. Хотя подобное использование не считается поддерживаемым, первоначальная причина изменения этих флагов не транзакционным образом давно неактуальна, поэтому можно исправить и вытекающее из неё решение.
- Недопущение повреждения записей в кеше планов в случаях, когда в кешированном плане оказывается команда CREATE DOMAIN или ALTER DOMAIN (Том Лейн)
- Отображение последних команд репликации, выполняемых передатчиками WAL, в представлении pg_stat_activity (Том Лейн)

  Ранее процесс walsender показывал свою последнюю SQL-команду, что вводило в заблуждение, если фактически он выполнял репликационную операцию. Теперь на том же основании, что и команды SQL, выводятся команды протокола репликации.
- Отображение в поле pg_settings.pending_restart значения true в случае удаления соответствующей записи из postgresql.conf (Альваро Эррера)

  Ранее поле pending_restart корректно отражало ситуацию, когда изменяется запись, задающая параметр, который вступает в силу только после перезапуска сервера, но не ситуацию, когда такая запись полностью удаляется.
- Устранение ошибки в особом случае, когда новый ведомый не мог начать работу с новым ведущим (Дилип Кумар, Роберт Хаас)

  При редком стечении ряда обстоятельств ведомый сервер мог застопориться при попытке нагнать неправильную линию времени в WAL.
- Корректировка минимальной точки восстановления в ситуации, когда при воспроизведении из WAL прерывания транзакции происходит усечение файла (Фудзии Масао)

  Усечение файла невозможно отменить, поэтому остановить восстановление в точке, предшествующей данной записи, в общем случае нельзя. Подобная ситуация с фиксированием транзакции была урегулирована много лет назад, но данная была упущена.

- Пресечение попыток поиска в каталоге при возникновении ошибки в приёмнике WAL (Масахико Савада, Бхарат Рупиредди)
- Обеспечение корректной реакции ведомого сервера, при запуске ожидающего поступления WAL, на команду отключения (Фудзии Масао, Соумйадип Чакраборти)
- Добавление блокировки во избежание чтения некорректных данных из файла relmapper при одновременной записи в него со стороны другого процесса (Хейкки Линнакангас)
- Усовершенствование проверок, выявляющих нарушения в сообщениях протокола репликации (Том Лейн)

  В рабочих процессах логической репликации часто использовались проверочные утверждения для выявления ситуаций, которые могли возникать при получении неправильных или перепутанных команд репликации. Такое решение видится не очень продуманным, поэтому данные проверки заменены на обычные, выдающие ошибки.
- Ликвидация дефектов и утечек памяти при логическом декодировании спекулятивного добавления (Дилип Кумар)
- Предотвращение смещения в счётчике использований плана в кеше при возникновении определённых ошибок в `CREATE TABLE ... AS EXECUTE` (Том Лейн)
- Устранение условий гонки, возможных при освобождении структур BackgroundWorkerSlot (Том Лейн)

  По всей вероятности, это не было причиной какой-либо из наблюдаемых ошибок на платформе Intel, но на платформах с менее строгими правилами обращений к памяти могли быть проблемы.
- Устранение возможности краха в коде сортировки (Ронан Данклау)

  Один путь в коде допускал попытку освободить нулевой указатель. При использовании сортировки ядром сервера этот путь был закрыт, но не исключено, что его могли открыть расширения.
- Предотвращение бесконечного цикла при добавлении данных в индекс SP-GiST (Том Лейн)

  В случае, когда неключевые столбцы (INCLUDE) занимают настолько много места, что листовой индексный кортеж в принципе не может поместиться в странице, класс операторов text_ops входил в бесконечный цикл, тщетно пытаясь уместить этот кортеж. Хотя в версиях до 11 индексы не поддерживали столбцы INCLUDE, эта защита от зацикливания перенесена и в старые версии, так как она позволяет также противостоять ошибкам в классах операторов.
- Обеспечение возможности прервать добавление данных в индекс SP-GiST сигналом отмены запроса (Том Лейн, Альваро Эррера)
- Ликвидация использования неинициализированной переменной, в результате которого реализация PL/pgSQL могла ошибочно полагать, что с предложением `INTO` указывалось `STRICT` (Том Лейн)
- Отказ от аварийного прерывания процесса в случае нехватки памяти в функции печати строк в libpq (Том Лейн)
- Допущение в ecpg возможности приведения значения `numeric`, равного INT_MIN (обычно -2147483648), к int (Джон Нейлор)
- Предотвращение в psql и других клиентских программах выхода за конец строки при обработке некорректно кодированных данных (Том Лейн)

  В случае нахождения у конца строки многобайтового символа, не соответствующего кодировке, различные циклы обработки могли выйти за завершающий строку NUL, что могло обойтись без последствий либо привести к краху программы, в зависимости от следующего содержимого памяти. Это отголоски CVE-2006-2313, хотя конкретно эти случаи не должны иметь интересных проявлений с точки зрения безопасности.

- Устранение предупреждений «invalid creation date in header» (неправильная дата создания в заголовке), возникающих при выполнении pg_restore с архивом, созданным в другом часовом поясе (Том Лейн)
- Добавление в pg_upgrade переноса значения `oldestXID` из старой инсталляции (Бертран Друво)

  Ранее в новых инсталляциях значение `oldestXID` обычно оказывалось достаточно старым для того, чтобы вызвать безусловную немедленную автоочистку для предотвращения зацикливания. Это нежелательно с точки зрения производительности, но что ещё хуже, инсталляции с большими значениями `autovacuum_freeze_max_age` могли экстренно останавливаться вскоре после обновления.
- Усовершенствование pg_upgrade для выявления расширений, которые следует обновить, и предложения дальнейших действий (Брюс Момджян)

  Теперь формируется скрипт, содержащий команды `ALTER EXTENSION UPDATE` для обновления расширений до версий, которые устанавливаются по умолчанию в новой инсталляции.
- Предупреждение проблем при переключении pg_receivewal между сжатым и несжатым форматом WAL (Микаэль Пакье)
- Пресечение попыток поиска в каталоге при возникновении ошибки в `contrib/postgres_fdw` (Том Лейн)

  Хотя обычно это работало, такой поиск может быть опасным, так как ошибке могла сопутствовать и потеря функциональности доступа к каталогу. В качестве побочного эффекта исправления теперь в сообщениях об ошибках преобразования данных упоминаются псевдонимы (если они заданы) столбцов и таблицы, фигурирующие в запросе, а не фактические имена нижележащей сторонней таблицы или столбцов.
- Улучшение инфраструктуры изоляционных тестов (Том Лейн, Микаэль Пакье)

  Добавлена возможность дополнить описание шагов в тесте указаниями ожидаемого порядка их выполнения. Это позволяет получать стабильные результаты (ранее результаты могли меняться в условиях гонки), не добавляя для исключения гонки длительные задержки, которые мы вставляли ранее (не вполне успешно). В качестве имён шагов/сеансов в тесте теперь допускаются идентификаторы без кавычек (ранее все такие имена должны были заключаться в кавычки). Также улучшен вывод результатов запросов в изоляционных тестах. Помимо этого, ликвидирован режим «dry-run» программы isolationtester. Наконец, в ней устранены утечки памяти.
- Уменьшение издержек при тестировании с затиранием кеша (Tom Lane)
- Исправление регрессионных тестов PL/Python для совместимости с Python 3.10 (Хонза Хорак)
- Изменение поведения вызова `printf("%s", NULL)`, который теперь должен выдавать `(null)`, а не останавливать сервер (Том Лейн)

  Это должно повысить устойчивость сервера в особых случаях, и при этом поведение `printf` в нашей реализации становится таким же, как в распространённых библиотеках.
- Исправление ошибочного сообщения в журнале, выдаваемого, когда восстановление на момент времени (PITR) останавливается на записи `ROLLBACK PREPARED` (Саймон Риггс)
- Уточнение в сообщениях об ошибках формулировки «неотрицательные значения» (Бхарат Рупиредди)
- Усовершенствование configure для работы с OpenLDAP версии 2.5, в которой теперь нет отдельной библиотеки `libldap_r` (Адриан Хо, Том Лейн)

  В случае отсутствия библиотеки `libldap_r` теперь неявно предполагается, что библиотека `libldap` является потокобезопасной.
- Добавление новых целей сборки `world-bin` и `install-world-bin` (Эндрю Дунстан)

Эти цели действуют так же, как `world` и `install-world`, соответственно, за исключением того, что они не собирают и не устанавливают документацию.

- Исправление правила make для TAP-тестов (`prove_installcheck`) с целью обеспечения их использования в PGXS (Эндрю Дунстан)
- Обеспечение возможности сборки 10 версии PostgreSQL с ICU 69 и новее (Питер Эйзентраут)
- Отказ от предположения, что строки, возвращаемые библиотеками GSSAPI, завершаются нулевым символом (Том Лейн)

  В спецификации GSSAPI строка задаётся указателем и длиной. На практике следующий за концом строки байт обычно нулевой, поэтому наш код раньше не сталкивался с проблемами; однако нам сообщили, что на это среагировал AddressSanitizer.
- Обеспечение возможности сборки с GSSAPI в среде MSVC (Микаэль Пакье)

  Устранение разнообразных несовместимостей с современными сборками Kerberos.
- Добавление для сборок MSVC параметра `--with-pgport` в набор параметров configure, которые выдаёт pg_config, в случае использования данного параметра (Эндрю Дунстан)

E.33. Выпуск 10.17

Дата выпуска: 2021-05-13

В этот выпуск вошли различные исправления, внесённые после версии 10.16. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.33.1. Миграция на версию 10.17

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Если вы обновляете сервер с более ранней версии, чем 10.16, см. также Раздел E.34.

E.33.2. Изменения

- Предотвращение целочисленного переполнения при вычислении индексов в массивах (Том Лейн)

  Ранее код обработки массивов не отслеживал переполнение целого при сложении нижней границы с длиной массива. В результате элементы с запредельными индексами (которые нельзя записать в виде целого числа) оказывались недоступными, но что ещё хуже, в дальнейшем они нарушали работу операций присваивания. Это могло приводить к перезаписи памяти и, как следствие, сбоям или нежелательным изменениям данных. (CVE-2021-32027)
- Исправление некорректной обработки «отбросовых» столбцов в целевых списках `INSERT ... ON CONFLICT ... UPDATE` (Том Лейн)

  Если список `UPDATE` содержал вложенные выборки, возвращающие несколько столбцов (в числе которых помимо столбцов с нужными данными могут быть отбросовые столбцы), выполнение `UPDATE` завершалось сохранением кортежей, включающих значения этих дополнительных столбцов. Это относительно безвредно само по себе, но если в таблицу добавлялись новые столбцы, эти значения оказывались недоступными, а если их типы данных отличались от типов добавленных столбцов, могли происходить сбои.

  Помимо этого, в версиях, которые поддерживают изменение, затрагивающее несколько секций, межсекционное изменение, вызванное в таком случае, порождало противоположную проблему: отбросовые столбцы удалялись из целевого списка, что обычно немедленно приводило к краху из-за нарушения в работе механизма вложенной выборки с несколькими столбцами. (CVE-2021-32028)

- Запрещение пометки допустимости NULL для столбца идентификации (Вик Фиринг)

  Указание `GENERATED ... AS IDENTITY` подразумевает ограничение `NOT NULL`, поэтому оно не должно приниматься с явным указанием `NULL`.

- Реализация возможности задавать в `ALTER ROLE/DATABASE ... SET` произвольные параметры `role`, `session_authorization` и `temp_buffers` (Том Лейн)

  Ранее излишне строгие проверки могли не допускать команды с некоторыми значениями, которые могли бы вполне корректно работать позже. Это создавало проблемы с упорядочиванием команд при выгрузке/загрузке данных и при обновлении.

- Устранение ошибки, связанной с приведением результата выражения `COLLATE` к типу, не поддерживающему сортировку (Том Лейн)

  При таком приведении строилось дерево разбора, в котором `COLLATE` должно было применяться к значению, не поддерживающему сортировку. Хотя обычно это ни на что не влияло (так как `COLLATE` не обрабатывается во время выполнения), но полученное с подобным приведением представление нельзя было загрузить после выгрузки.

- Недопущение вызова оконных функций и процедур через сообщения по протоколу «быстрого пути» (Том Лейн)

  На этом уровне поддерживаются только простые функции. При вызове агрегатной функции ошибка выдавалась и раньше, но при вызове оконной функции происходил сбой, а вызов процедуры мог выполниться успешно, только если процедура не манипулировала транзакциями.

- Добавление поддержки событийных триггеров в `pg_identify_object_as_address()` (Джоэл Джейкобсон)

- Исправление в функции `to_char()` обработки кодов формата, задающих месяц римскими цифрами, с отрицательными интервалами (Жюльен Руо)

  Ранее при вызове функции с такими параметрами обычно происходил сбой.

- Проверка указания в аргументе функции `pg_import_system_collations()` корректного OID схемы (Tom Lane)

- Устранение обращения к неинициализированному значению при разборе квантификатора `\{`*m*`,`*n*`\}` в регулярном выражении в режиме BRE (Том Лейн)

  В результате ошибки этот квантификатор мог работать как «нежадный», то есть как квантификатор `{`*m*`,`*n*`}?` в расширенных регулярных выражениях.

- Рассмотрение системных столбцов при оценивании количества групп с использованием расширенной статистики (Томаш Вондра)

  Ранее для запросов вида `SELECT ... GROUP BY a, b, ctid` могли выдаваться странные оценки.

- Предотвращение деления на ноль при оценивании избирательности регулярного выражения с очень длинным фиксированным префиксом (Том Лейн)

  При таком условии в качестве значения избирательности выдавалось `NaN`, что приводило к сбоям проверочных утверждений или непонятному поведению планировщика.

- Устранение ошибки выхода за границу таблицы при сканировании битовой карты индекса BRIN (Томаш Вондра)

  При использовании в индексе BRIN страничных зон размера, не равного степени двух, были возможны особые случаи, когда в ходе сканирования битовой карты происходило обращение к странице за физическим концом таблицы, приводившее к ошибке «could not open file» (не удалось открыть файл).

- Недопущение некорректной смены линии времени при восстановлении незафиксированных двухфазных транзакций из WAL (Соумйадип Чакраборти, Джимми Йи, Кевин Йеп)

  Вследствие исправленной теперь ошибки последующие записи WAL сохранялись с идентификатором неправильной линии времени, что порождало проблемы с согласованностью данных и даже могло привести к остановке сервера при последующем перезапуске.

- Освобождение всех блокировок в случае остановки стартового процесса ведомого сервера (Фудзии Масао)

  В случае остановки ведомого сервера в ходе восстановления некоторые блокировки могли сохраниться. Это приводило к сбоям проверочных утверждений в отладочных сборках; могло ли это иметь серьёзные последствия в обычных сборках, неизвестно.

- Устранение сбоя при выполнении команды `ALTER SUBSCRIPTION REFRESH` в рабочем процессе логической репликации (Питер Смит)

  Код ядра не выполняет такую команду, но она вполне может выполняться в триггере реплики.

- Использование по умолчанию значения `wal_sync_method` = `fdatasync` в последних версиях FreeBSD (Томас Манро)

  Во FreeBSD 13 поддерживается режим `open_datasync`, и он в принципе должен использоваться по умолчанию, однако пока не ясно, в чём его преимущество для Postgres, решено оставить прежний вариант.

- Обеспечение полной очистки в случае прерывания при отсоединении сегмента DSM (Томас Манро)

  В результате исправленной теперь ошибки временные файлы могли не очищаться должным образом после параллельного запроса.

- Устранение утечки памяти при инициализации SSL-параметров сервера (Микаэль Пакье)

  Объём утечки обычно был незначительным, но он мог увеличиваться, если главному процессу неоднократно посылались сигналы SIGHUP.

- Устранение разнообразных незначительных утечек памяти в коде сервера (Том Лейн, Андрес Фройнд)

- Предотвращение бесконечного цикла в libpq при получении некорректной длины в сообщении ParameterDescription (Том Лейн)

- Использование правильных разделителей (обратной косой черты) в пути к программе pg_ctl, который выводит initdb в инструкции по запуску сервера (Нитин Ядав)

- Восстановление в psql предыдущего поведения присваивания `\connect service=`*`значение`* (Tom Lane)

  В результате предыдущего исправления ошибки переменные окружения (например, `PGPORT`) стали переопределять записи в файле служб в данном контексте. Сейчас восстановлено прежнее поведение, при котором действует обратный приоритет.

- Устранение условий гонки при обнаружении изменений в файле командой psql `\e` и связанными с ней командами (Лауренц Альбе)

  При очень быстром редактировании файла можно было ввести в заблуждение проверку изменения временного файла, отслеживающую время изменения файла.

- Добавление недостающей проверки версии файла в pg_restore (Том Лейн)

  При чтении архива в специальном формате из источника, не поддерживающего позиционирование, утилита pg_restore забывала проверить версию архива. В случае загрузки архива более новой версии, чем она поддерживает, впоследствии могла возникнуть путаница.

- Добавление в pg_upgrade дополнительных проверок на наличие в пользовательских таблицах типов, не поддерживающих обновление (Том Лейн)

  Усовершенствование выявления случаев, когда тип, не поддерживающий обновление, оказывается внутри типа-контейнера (например, в типе массива или диапазона). Также теперь запрещается обновление, если пользовательские таблицы содержат системные составные типа, так как OID таких типов может меняться от версии к версии.

- Исправление в pg_waldump подсчёта записей `XACT` при сборе статистики по записям (Кётаро Хоригути)

- Ликвидация в `contrib/amcheck` ошибочного проверочного утверждения, не допускающего одновременно установленные в кортеже флаги `HEAP_XMAX_LOCK_ONLY` и `HEAP_KEYS_UPDATED` (Жюльен Руо)

  Такое состояние вполне легально после `SELECT FOR UPDATE`.

- Корректировка правил сборки с VPATH для совместимости с последними версиями компилятора Oracle Developer Studio (Ной Миш)

- Исправление тестирования языка PL/Python для Python 3 в ОС Solaris (Ной Миш)

E.34. Выпуск 10.16

Дата выпуска: 2021-02-11

В этот выпуск вошли различные исправления, внесённые после версии 10.15. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.34.1. Миграция на версию 10.16

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Однако обратите внимание на первый пункт в списке изменений, описывающий случаи, в которых рекомендуется перестроить индексы после обновления версии.

Также, если вы обновляете сервер с более ранней версии, чем 10.11, см. Раздел E.39.

E.34.2. Изменения

- Добавление в `CREATE INDEX CONCURRENTLY` ожидания завершения подготовленных транзакций (Андрей Бородин)

  На этапе, когда команда `CREATE INDEX CONCURRENTLY` ждёт завершения всех выполняющихся в это время транзакций, чтобы она могла увидеть добавленные ими строки, она с той же целью должна дожидаться завершения и всех подготовленных транзакций. В противном случае строки, добавленные подготовленными транзакциями, могут не попасть в новый индекс, вследствие чего запросы, использующие этот индекс, не будут их видеть. В инсталляциях, где включены подготовленные транзакции (`max_prepared_transactions` > 0), рекомендуется перестроить все ранее построенные в неблокирующем режиме индексы, если их могла затронуть эта проблема.

- Недопущение ошибочных результатов при применении `WHERE CURRENT OF` к курсору, план выполнения которого содержит узел MergeAppend (Том Лейн)

  Такая конструкция не поддерживается (вообще говоря, курсор с `ORDER BY` не обязательно будет поддерживать простое изменение); ранее её не запрещалось использовать, но это было чревато незаметными ошибками.

- Устранение сбоя при применении `WHERE CURRENT OF` к курсору, план выполнения которого содержит нестандартный узел сканирования (Дэвид Гейер)

- Исправление обработки местозаполнителя, который вычисляется на некотором уровне соединения и используется только на этом же уровне (Том Лейн)

  В результате данного упущения планировщик мог выдавать ошибки «failed to build any *N*-way joins» (не удалось построить никакие *N*-сторонние соединения).

- Проверка поддержки пометки/восстановления позиций индексными методами доступа (Эндрю Гирт)

  Тем самым предупреждаются ошибки с сообщениями об отсутствии опорных функций, возникавшие в редких особых случаях.

- Изменение параметров для решения проблемы нехватки слотов DSM при очень активном использовании параллельных запросов (Томас Манро)

- Обеспечение корректной обработки в команде `ALTER DEFAULT PRIVILEGES` повторяющихся аргументов (Микаэль Пакье)

  Ранее в случае повторения имени роли или схемы в одной команде могли выдаваться ошибки «tuple already updated by self» (кортеж уже изменился сам собой) или нарушаться ограничения уникальности.

- Осуществление сброса связанных с ACL кешей при изменениях в `pg_authid` (Ной Миш)

  Тем самым гарантируется, что решения, связанные с правами доступа, будут приниматься с учётом действия `ALTER ROLE ... [NO] INHERIT`.

- Предотвращение некорректной обработки неоднозначных предложений `CREATE TABLE LIKE` (Том Лейн)

  Предложение `LIKE` повторно рассматривается после создания новой таблицы, когда выполняется импорт индексов и т. п. Ранее при повторном анализе могла быть найдена другая таблица с тем же именем, что приводило к неожиданному поведению; например, это происходило при создании новой временной таблицы с тем же именем, что у исходной таблицы в `LIKE`.

- Изменение порядка действий в `CREATE TABLE LIKE`, чтобы индексы копировались до создания ограничений внешнего ключа (Том Лейн)

  Теперь случай, когда ограничение внешнего ключа, объявленное на внешнем уровне `CREATE TABLE` и ссылающееся на саму таблицу, зависит от индекса, создаваемого предложением `LIKE`, обрабатывается корректно.

- Запрет выполнения `CREATE STATISTICS` с системными каталогами (Томаш Вондра)

- Недопущение преобразования наследуемой дочерней таблицы в представление (Том Лейн)

- Обеспечение корректного освобождения дискового пространства, выделенного для удаляемого отношения, при фиксировании транзакции (Томас Манро)

  Ранее, если удаляемое отношение занимало несколько сегментов по 1 ГБ, немедленно освобождался объём только первого. Файлы остальных сегментов просто удалялись, что не позволяло ядру освободить занимаемое ими место, пока эти файлы были открытыми в других обслуживающих процессах.

- Исправление обработки многобайтовых символов, экранированных обратной косой чертой, в `COPY FROM` (Хейкки Линнакангас)

  Ранее многобайтовый символ, идущий после обратной косой черты, обрабатывался некорректно. Так, в некоторых клиентских кодировках часть многобайтового символа могла восприниматься как разделитель полей или как маркер конца данных.

- Отказ от ненужного создания хеш-таблиц для исполнителя при выполнении `EXPLAIN` без `ANALYZE` (Алексей Баштанов)

- Устранение недавно привнесённых условий гонки при обработке очереди LISTEN/NOTIFY (Том Лейн)

  Только что подключившийся к очереди процесс-приёмник уведомлений мог попытаться прочитать страницы SLRU, которые в этот момент обрезались, что могло привести к ошибке.

  Указатель на хвост очереди мог получить значение, не соответствующее позиции ни одного из процессов, в результате чего логика очистки очереди фактически отключалась. Продолжающиеся вызовы NOTIFY затем приводили к появлению предупреждений о переполнении очереди, и в конце концов передача уведомлений полностью останавливалась до перезапуска сервера.

- Реализация поддержки всех возможных сочетаний типов JSON оператором конкатенации jsonb (Том Лейн)

  До этого поддерживалась конкатенация двух объектов или двух массивов JSON. Теперь обрабатываются и другие случаи — отличные от массивов аргументы помещаются в одноэлементные массивы, которые затем складываются. Ранее некоторые сочетания аргументов обрабатывались по этому принципу, а с другими возникали ошибки.

- Исправление ошибки обращения к неинициализированному значению при разборе квантификатора * в регулярном выражении в режиме BRE (Том Лейн)

  В результате ошибки этот квантификатор мог работать как «нежадный», то есть как квантификатор *? в расширенных регулярных выражениях.

- Исправления поведения power() со значением numeric, равным INT_MIN (-2147483648) (Дин Рашид)

  Ранее для такого числа возвращался результат без значимых цифр.

- Предотвращение возможной потери данных вследствие некорректного определения позиции зацикливания в журнале SLRU (Ной Миш)

  Позиция зацикливания обычно оказывается в середине страницы и должна округляться до границы страницы, однако это делалось неправильно. Негативные последствия это могло иметь, только если SLRU-кеш переполнялся в пределах одной страницы, что маловероятно в правильно функционирующей системе. Проявляться это могло в последующих ошибках «apparent wraparound» (видимо, произошло зацикливание) и «could not access status of transaction» (не удалось получить состояние транзакции).

- Устранение утечек памяти в процессах walsender при передаче новых снимков для логического декодирования (Амит Капила)

- Исправление поведения walsender при получении дополнительных команд после завершения репликации (Джефф Девис)

- Выявление взаимоблокировок, возможных между серверами горячего резерва и стартовым процессом (применяющим WAL) (Фудзии Масао)

  Стартовый процесс не выполнял процедуру обнаружения взаимоблокировок, поэтому когда он оказывался последним в ситуации взаимного ожидания, взаимоблокировка не диагностировалась.

- Обеспечение очистки необработанных запросов к фоновым рабочим процессам в начале процедуры выключения в режимах «smart» и «fast» (Том Лейн)

  Ранее были возможны условия гонки, когда дочерний процесс, запросивший для себя фоновый рабочий процесс непосредственно перед выключением сервера, оказывался в состоянии бесконечного ожидания и таким образом препятствовал выключению.

- Устранение платформенных особенностей при разборе значений recovery_target_xid (Микаэль Пакье)

Целевой XID может задаваться как 64-битное значение, а для его разбора использовалась функция `strtoul()`, неподходящая для этого на платформах, где `long` имеет размер 32 бита (например, в Windows).

- Устранение сбоя утверждения истинности в `pg_get_functiondef()` при анализе функции с указанием `TRANSFORM` (Том Лейн)
- Восстановление в psql возможности передавать пароль в аргументе *`строка_соединения`* команды `\connect` (Том Лейн)

  Это работало раньше, но в результате недавнего исправления этот пароль перестал восприниматься (и поэтому запрашивался дополнительно).
- Устранение разнообразных дефектов в реализации команды psql `\help` (Кётаро Хоригути, Том Лейн)

  Команда `\help` с двумя словами в аргументах не могла найти описание команды только для первого слова. Например, команда `\help reset all` должна была показать справку по `RESET`, но она этого не делала. Кроме того, `\help` в ряде случаев не вызывала постраничник, тогда как должна была. Наконец, в прежней реализации имелись утечки памяти.
- Изменение поведения pg_dump с тем, чтобы в скрипте восстановления команды `ALTER PUBLICATION ADD TABLE` выполнялись от имени владельца публикации (Том Лейн)

  Ранее эти команды выполнялись от имени роли, запустившей скрипт восстановления; в большинстве случаев это должно работать, но в некоторых ситуациях эта роль может не иметь нужных прав.
- Исправление в pg_dump обработки указаний `WITH GRANT OPTION` в изначально устанавливаемых правах в расширениях (Ной Миш)

  В случае, когда скрипт расширения создаёт объект и даёт права доступа к нему с правом передачи, а пользователь затем отзывает эти права, pg_dump не мог выдать корректный SQL восстановления состояния. (Эта проблема если и затрагивает, то лишь немногие расширения.)
- Рассмотрение в процедуре pg_rewind всех требующихся WAL-файлов при синхронизации резервного сервера (Иэн Барвик, Хейкки Линнакангас)
- Передача имени корректной базы данных в сообщениях об ошибках подключения, выдаваемых некоторых клиентскими программами (Альваро Эррера)

  Когда имя базы данных не задавалось в командной строке, а выбиралось подразумеваемое по умолчанию, программы pg_dumpall, pgbench, oid2name и vacuumlo выдавали неверные сообщения об ошибках в случае сбоя подключения.
- Устранение утечки памяти в `contrib/auto_explain` (Ли Япинь)

  Память, занятая в процессе формирования вывода `EXPLAIN`, не освобождалась до конца текущей транзакции (для оператора верхнего уровня) или до завершения внешнего оператора (для вложенного оператора). В частности это создавало проблемы при включённом параметре `log_nested_statements`.
- Устранение в `contrib/postgres_fdw` утечки открытых подключений к внешним серверам при удалении сопоставлений пользователей или объекта стороннего сервера (Бхарат Рупиредди)

  Использовать подключения, связанные с удалённым сопоставлением пользователя или сторонним сервером, нельзя, но ранее они сохранялись открытыми на протяжении локального сеанса.
- Добавление в `contrib/pgcrypto` проверки кода возврата, выдаваемого функциями OpenSSL EVP (Микаэль Пакье)

В действительности ошибки в этом месте не ожидаются, но это изменение убирает предупреждения статистических анализаторов.

- Предотвращения сбоя в коде поддержки индексов GiST в модуле `contrib/pg_trgm`, происходившего в редком случае вызова функции picksplit для разделения ровно двух элементов (Эндрю Гирт, Александр Коротков)
- Исправление расчёта тайм-аутов в `contrib/pg_prewarm` и `contrib/postgres_fdw` (Алексей Кондратов, Том Лейн)

  В основном цикле главного процесса `contrib/pg_prewarm` ошибочно вычислялась задержка в 1000 раз меньше ожидаемой, вследствие чего он создавал лишнюю нагрузку для процессора. Ожидая результата от удалённого сервера, модуль `contrib/postgres_fdw` устанавливал вместо желаемого тайм-аута в 1000 раз больший (хотя эта ошибка нивелировалась 60-секундным ограничением сверху).

  Обе эти ошибки были следствием неправильного пересчёта секунд с микросекундами в миллисекунды. Чтобы в будущем таких ошибок не было, добавлена удобная функция `TimestampDifferenceMilliseconds()`.
- Улучшение логики configure в части выбора `PG_SYSROOT` в macOS (Том Лейн)

  Новый подход с большей вероятностью даст желаемый результат в случаях, когда Xcode новее нижележащей операционной системы. В случае выбора sysroot, не соответствующего версии ОС, скомпилированные программы могут не работать.
- Добавление при сборке в macOS ключа `-isysroot` на этапах компоновки и компиляции (Джеймс Хиллиард)

  Это должно помогать в ситуациях, когда версия Xcode не согласуется с версией операционной системы.
- Обновление данных часовых поясов до версии tzdata 2021a, включающее изменение правил перехода на летнее время в России (смену часового пояса в Волгограде) и Южном Судане, а также корректировку исторических данных для Австралии, Багам, Белиза, Бермуд, Ганы, Израиля, Кении, Нигерии, Палестины, Сейшел и Вануату.

  В частности, часовой пояс Australia/Currie был признан равнозначным Australia/Hobart и ненужным.

E.35. Выпуск 10.15

Дата выпуска: 2020-11-12

В этот выпуск вошли различные исправления, внесённые после версии 10.14. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.35.1. Миграция на версию 10.15

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Если вы обновляете сервер с более ранней версии, чем 10.11, см. также Раздел E.39.

E.35.2. Изменения

- Запрещение конструкции `DECLARE CURSOR ... WITH HOLD` и вызова отложенных триггеров в выражениях индексов и запросов матпредставлений (Ной Миш)

  Устранённая уязвимость по сути позволяла выйти из защищённой среды, в которой выполняются «операции с ограничениями безопасности». Злоумышленник, имеющий право создания не временных SQL-объектов, мог воспользоваться этой уязвимостью для выполнения произвольного SQL-кода от имени суперпользователя.

Проект PostgreSQL благодарит Этьена Сталманса за сообщение об этой проблеме. (CVE-2020-25695)

- Исправление обработки сложных параметров в строках подключений в программах pg_dump, pg_restore, clusterdb, reindexdb и vacuumdb (Том Лейн)

  В аргументе `-d` программ pg_dump и pg_restore, а также в аргументе `--maintenance-db` других перечисленных программ может задаваться «строка подключения», содержащая не просто имя базы данных, но и другие параметры подключений. В случаях, когда эти программы должны были устанавливать дополнительные соединения, например, для параллельной обработки или работы с несколькими базами, строка соединения терялась, и для дополнительных подключений использовались только основные параметры (адрес и порт сервера, имя базы данных и имя пользователя). Это могло приводить к сбоям подключений, если строка подключения дополнительно содержала существенную информацию, например, изменённые параметры SSL или GSS. Хуже того, успешно установленное соединение могло оказаться незашифрованным (тогда как должно было шифроваться) или подверженным атакам с посредником (которые предотвращались бы при использовании заданных параметров). (CVE-2020-25694)

- Обеспечение использования всех не переопределённых параметров из предыдущей строки соединения при выполнении команды psql `\connect` (Том Лейн)

  Это исправление предотвращает ошибки переподключения, которые могли раньше возникать из-за упущения значимых параметров, например, параметров SSL или GSS. Хуже того, раньше переподключение могло произойти успешно, но при этом оказаться незашифрованным (тогда как должно было шифроваться) или подверженным атакам с посредником (которые предотвращались бы при использовании заданных параметров). По большому счёту это та же проблема, что и описанная выше проблема pg_dump и др., но в случае psql ситуация усложняется тем, что пользователь может намеренно переопределить некоторые параметры соединений. (CVE-2020-25694)

- Недопущение изменения командой psql `\gset` специальных переменных (Ной Миш)

  Команда `\gset` без префикса ранее переопределяла любые переменные по указанию сервера. Таким образом, скомпрометированный сервер мог установить переменную специального назначения, например `PROMPT1`, в результате чего было возможно выполнение произвольного кода оболочки в сеансе пользователя.

  Проект PostgreSQL благодарит Ника Клитона за сообщение об этой проблеме. (CVE-2020-25696)

- Предотвращение возможной потери данных при параллельном усечении журналов SLRU (Ной Миш)

  Из-за этого редко проявлявшегося дефекта впоследствии могли выдаваться ошибки «apparent wraparound» (видимо, произошло зацикливание) или «could not access status of transaction» (не удалось получить состояние транзакции).

- Обеспечение должного сохранения каталогов SLRU на диске во время контрольных точек (Томас Манро)

  Тем самым предупреждается риск потери данных в случае последующего сбоя операционной системы.

- Исправление поведения `ALTER ROLE` для пользователей, имеющих атрибут `BYPASSRLS` (Том Лейн, Стивен Фрост)

  Атрибут `BYPASSRLS` у ролей могут изменять только суперпользователи, но другие действия `ALTER ROLE`, например смена пароля, должны разрешаться и непривилегированным пользователям с учётом обычных ограничений. Предыдущая реализация разрешала любые операции с ролью, имеющей такой атрибут, только суперпользователям.

- Исправление обработки выражений `CREATE TABLE LIKE` с наследованием (Том Лейн)

  Если команда `CREATE TABLE` использует и традиционное наследование, и `LIKE`, ссылки на столбцы в ограничениях `CHECK` и выражениях индексов, приходящие из родительской таблицы `LIKE`, могли быть неправильно пронумерованы, вследствие чего выдавались некорректные результаты или странные сообщения об ошибках. То же самое могло наблюдаться с генерируемыми выражениями (в тех версиях, где они поддерживаются).

- Исправление ошибки со сдвигом на один отрицательных значений, соответствующих годам до н. э. в функциях `to_date()` и `to_timestamp()` (Дар Альатар-Йемен, Том Лейн)

  Кроме того, отрицательное значение года с явным указанием «BC» (до н. э.) теперь становится положительным, относящимся к н. э.

- Обеспечение архивирования файлов истории линий времени вместе с WAL на ведомых серверах при выбранном в `archive_mode` режиме `always` (Григорий Смолкин, Фудзии Масао)

  Вследствие устранённого теперь упущения можно было лишиться возможности восстановления на момент времени (PITR).

- Устранение ошибки «cache lookup failed for relation 0» (ошибка поиска в кеше для отношения 0) в рабочих процессах логической репликации (Том Лейн)

  Практические следствия этой ошибки незначительны, так как она весьма маловероятна, а в случае её возникновения рабочий процесс просто перезапускается.

- Предотвращение передачи избыточных контрольных запросов процессами логической репликации (Том Лейн)

- Выполнение в процедуре отключения «smart» остановки фоновых процессов только после завершения всех клиентских сеансов (сеансов переднего плана) (Том Лейн)

  С прежним поведением нарушалось выполнение параллельных запросов, так как управляющий процесс принудительно завершал процессы параллельных исполнителей и не запускал новые. Также при этом переставала работать автоочистка, что могло быть чревато большими проблемами, если продолжавшие существование клиентские сеансы вносили множество изменений данных.

- Предупреждение переполнения стека при обработке сигналов в процессе postmaster (Том Лейн)

  При активной параллельной обработке наблюдались сбои процесса postmaster по причине поступления слишком большого количества сигналов, запрашивающих создание параллельных исполнителей.

- Недопущение вызова обработчиков atexit при выходе по сигналу SIGQUIT (Кётаро Хоригути, Том Лейн)

  Большинство серверных процессов уже отказались от этой практики, но процесс архиватора по недосмотру сохранил старое поведение. Также неправильно завершались процессы, находившиеся в состоянии ожидания стартового клиентского пакета.

- Предотвращение неправильной оптимизации ограничений подзапроса, обращающихся к группирующим столбцам, которые представляются константами (Том Лейн)

  «Константный» выходной столбец подзапроса в действительности не будет константой, если это группирующий столбец, фигурирующий только в некоторых из наборов группирования.

- Предупреждение сбоя в случаях, когда внедрённая SQL-функция меняет форму потенциально хешируемого выражения сравнения для подплана (Том Лейн)

- Корректировка поведения в случае появления новых HOT-цепочек в ходе построения или перестроения индекса, когда одновременно происходит изменение данных (Анастасия Лубенникова, Альваро Эррера)

В результате устранённого теперь упущения могли выдаваться ошибки «failed to find parent tuple for heap-only tuple» (не удалось найти родительский кортеж для HOT-кортежа).

- Устранение сбоя при параллельном сканировании B-дерева в случае, когда условие индекса не удовлетворяется (Джеймс Хантер)
- Обеспечение распаковки данных перед добавлением их в индекс BRIN (Томаш Вондра)

  В элементах индексов не должны содержаться указатели на внешние данные TOAST, однако в реализации BRIN это не учли. Вследствие этого могли возникать ошибки «missing chunk number 0 for toast value NNN» (отсутствует порция номер 0 для TOAST-значения NNN). (Если вы столкнулись с такой ошибкой в существующем индексе, для устранения её должно быть достаточно выполнить `REINDEX`.)
- Корректировка обработки сброса обобщения в случае, когда он осуществляется одновременно со сканированием индекса BRIN (Александр Лахин, Альваро Эррера)

  Ранее, если сброс обобщения зоны страниц происходил в неудачный момент времени, при сканировании индекса могла выдаваться ложная ошибка с сообщением о повреждении индекса.
- Устранение редкой ошибки «lost saved point in index» (потеряна сохранённая точка в индексе) при сканировании составных индексов GIN (Том Лейн)
- Исправление непереносимого использования `getnameinfo()` в реализации представления `pg_hba_file_rules` (Том Лейн)

  Из-за ликвидированной теперь ошибки на FreeBSD 11 и возможно других платформах столбцы `address` и `netmask` в этом представлении всегда содержали null.
- Устранение риска использования памяти после освобождения при отслеживании в событийном триггере операции `ALTER TABLE` (Жеан-Гийом де Рорте)
- Исправление некорректного сообщения об ошибке, выдаваемой при несовпадении типов движущегося агрегата (Джефф Джейнс)
- Устранение блокировки при получении от параллельного исполнителя очень большого сообщения об ошибке (Вигнеш Си)
- Избавление от неоправданной ошибки при передаче очень большого объёма данных через очереди в разделяемой памяти (Маркус Ваннер)
- Устранение утечек памяти в кеше отношений при использовании политик RLS (Том Лейн)
- Устранение небольшой утечки памяти при обработке сигнала SIGHUP, возникавшей с новыми значениями GUC-переменных, которые нельзя изменить без перезагрузки (Том Лейн)
- Поддержка в libpq строк произвольной длины в файлах `.pgpass` (Том Лейн)

  Это прежде всего полезно, когда в качестве паролей сохраняются очень длинные защищённые ключи.
- Отказ в libpq для Windows от вызовов `WSACleanup()` и сведение вызовов `WSAStartup()` к одному единственному для процесса (Том Лейн, Александр Лахин)

  Ранее libpq вызывала `WSAStartup()` при установлении каждого соединения, а `WSACleanup()` — при закрытии соединения. Однако как оказалось, вызов `WSACleanup()` может повлиять на другие операции программы, а именно, мы иногда наблюдали исчезновение ожидаемого вывода программы. По всей видимости, ликвидация этого вызова не должна иметь никаких побочных эффектов, поэтому он был удалён. (Это также способствует повышению производительности, если программа устанавливает множество подключений к серверу, так как она не будет постоянно загружать и выгружать DLL.)
- Исправление логики инициализации потоков в библиотеке ecpg для Windows (Том Лейн, Александр Лахин)

Из-за некорректной установки блокировок многопоточные приложения ecpg при редком стечении обстоятельств могли работать некорректно.

- Смена в Windows режима чтения вывода команды «обратный апостроф» в psql с двоичного на текстовый (Том Лейн)

  Благодаря этому будут корректно обрабатываться символы новой строки.

- Включение в pg_dump сбора информации о столбцах в таблицах конфигурации расширений (Фабрицио де Ройес Мелло, Том Лейн)

  Из-за отсутствия этой информации происходили сбои при загрузке данных таблицы, выгруженных с указанием `--inserts` или с неполными (хотя обычно корректными) командами `COPY`.

- Добавление в pg_upgrade проверки существования каталогов табличных пространств в целевом кластере (Брюс Момджян)

- Устранение возможности утечки памяти в `contrib/pgcrypto` (Микаэль Пакье)

- Добавление проверки на случай маловероятной ошибки во время выполнения в `contrib/pgcrypto` (Даниэль Густафссон)

- Исправление недавно добавленных в тесты проверок `timetz`, которые работали только когда в США действовало летнее время (Том Лейн)

- Обновление данных часовых поясов до версии tzdata 2020d, включающее изменение правил перехода на летнее время на Фиджи, в Марокко, Палестине, канадском Юконе, на острове Маккуори и на станции Кейси (Антарктика), а также корректировку исторических данных для Франции, Венгрии, Монако и Палестины.

- Синхронизация нашей копии библиотеки timezone с выпущенной IANA версией tzcode 2020d (Том Лейн)

  С этой версией к нам попало изменение выбираемого по умолчанию формата вывода zic с «fat» (расширенный) на «slim» (минимальный). Для наших целей это не имеет практического значения, так как мы продолжаем использовать формат «fat» в версиях ниже 13. Также теперь гарантируется, что `strftime()` будет менять значение `errno`, только если произойдёт ошибка.

E.36. Выпуск 10.14

Дата выпуска: 2020-08-13

В этот выпуск вошли различные исправления, внесённые после версии 10.13. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.36.1. Миграция на версию 10.14

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Если вы обновляете сервер с более ранней версии, чем 10.11, см. также Раздел E.39.

E.36.2. Изменения

- Назначение безопасного значения `search_path` в процессах, применяющих изменения, и процессах-передатчиках WAL, участвующих в логической репликации (Ной Миш)

  Злонамеренный пользователь, подключённый к базе подписчика или публикации, имел возможность запустить произвольный SQL-код от имени роли, осуществляющей репликацию (обычно это суперпользователь). Некоторые риски аналогичны ранее описанным в CVE-2018-1058, и в данном исправлении для их ликвидации приёмник и

передатчик, участвующие в логической репликации, теперь устанавливают пустое значение `search_path`. (Как и с CVE-2018-1058, это изменение может вызывать проблемы при использовании неполных имён в определениях реплицируемых таблиц.) Другие риски связаны с реплицированием объектов, принадлежащих недоверенным ролям; лучшее, что с этим можно сделать, — описать эти угрозы в документации. (CVE-2020-14349)

- Повышение уровня безопасности в установочных скриптах дополнительных модулей (Том Лейн)

  Атаки, подобные описанным в CVE-2018-1058, могли осуществляться и через установочный скрипт расширения, если злоумышленник мог создавать объекты либо в целевой схеме расширения, либо в схеме другого расширения, от которого зависит первое. Так как для установки расширения требуются права суперпользователя, это направление открывало для обычного пользователя возможность получения прав суперпользователя. Для устранения этого риска в установочных скриптах назначен безопасный путь `search_path`, отключён параметр `check_function_bodies` и исправлены имеющиеся в некоторых модулях в contrib запросы, модифицирующие каталог, в соответствии с требованиями безопасности. В документацию добавлена информация для разработчиков сторонних расширений, которая должна помочь им сделать свои установочные скрипты безопасными. Однако описанных мер может быть недостаточно — расширения, зависящие от других расширений, подвержены риску атаки при неосторожной установке. (CVE-2020-14350)

- Ликвидация ошибки в передатчике WAL, в результате которой он переставал передавать сообщения обратной связи после передачи сообщения об активности (Альваро Эррера)

  Это не создавало больших проблем в случае использования встроенной логической репликации, так как встроенный приёмник WAL всё равно достаточно часто посылает ответные сообщения (и тем самым сбрасывает некорректное состояние). Но с некоторыми другими системами репликации, например pglogical, последствия были более серьёзными.

- Исправление срабатывания триггеров `UPDATE`, привязанных к определённым столбцам, на стороне подписчиков логической репликации (Том Лейн)

  В прежней реализации не было учтено, что номера столбцов на стороне публикации и на стороне подписчика могут быть разными, и если они действительно различались, триггеры срабатывали некорректно.

- Устранение замедления `ts_headline()` (Том Лейн)

  Добавленное в предыдущем наборе корректирующих выпусков исправление фразового поиска могло провоцировать катастрофическое замедлению `ts_headline()` при обработке больших документов; мало того, запрос, выполнение которого заходило в проблематичный цикл, нельзя было отменить.

- Обеспечение возможности прерывания функции `repeat()` при отмене запроса (Джо Конвей)

- Добавление в `pg_current_logfile()` символа возврата каретки (`\r`) в результат, выдаваемый в Windows (Том Лейн)

- Корректировка обработки значений `NaN` при параллельном агрегировании данных по столбцам типа `numeric` (Том Лейн)

  Если некоторые рабочие процессы, выполняющие частичное агрегирование, находили в своих данных только `NaN`, а другие наоборот, `NaN` не находили, их результаты объединялись некорректно, вследствие чего мог быть некорректным и общий результат (то есть это не было ожидаемое значение `NaN`).

- Недопущение указания в качестве времени дня значений, превышающих 24 часа (Том Лейн)

  Код ввода даты/времени был намеренно написан так, чтобы этот тип принимал время «24:00:00» или равнозначное ему «23:59:60», но не большие значения. Однако проверка интервала была не очень точной, и в результате также допускались значения «23:59:60.*nnn*»

с ненулевой дробной частью секунд *nnn*. Таким образом, результирующее значение даты/времени оказывалось в первой секунде следующего дня. С типами же `time` и `timetz` сохранённые значения фактически выходили за 24 часа, вследствие чего могли иметь место проблемы при выгрузке/восстановлении данных и другие нежелательные эффекты.

- Отказ от заключения в двойные кавычки имён индексов в выходных форматах `EXPLAIN`, отличных от текстового (Том Лейн, Эйлер Тавейра)
- Исправление в `EXPLAIN` учёта использования ресурсов, в частности, обращений к буферам в параллельных исполнителях при выполнении плана с узлами `Gather Merge` (Жеан-Гийом де Рорте)
- Выбор другого момента для перепроверки ограничения в процедуре `ALTER TABLE` (Дэвид Роули)

  В некоторых случаях, когда при выполнении `ALTER TABLE` необходимо полностью переписать содержимое таблицы (например, потому что изменился тип данных столбца) и при этом просканировать таблицу, чтобы перепроверить внешние ключи или ограничения `CHECK`, действия выполнялись в неправильном порядке, что могло проявляться в странных ошибках вида «could not read block 0 in file "base/nnnnn/nnnnn": read only 0 of 8192 bytes» (не удалось прочитать блок 0 в файле "base/nnnnn/nnnnn" (прочитано байт: 0 из 8192)).
- Косвенное исправление некорректной пометки полей `pg_subscription.subslotname` и `pg_subscription_rel.srsublsn` как NOT NULL (Том Лейн)

  В данных начальной загрузки каталога эти два поля в каталоге были некорректно помечены как не принимающие NULL. В существующих инсталляциях исправить эту ошибку простым способом нельзя (тогда как для версии 13 и последующих пометка скорректирована). Корректность этих пометок оказалась важна прежде всего в JIT-процедуре разбора кортежей, в неё и пришлось добавить явное переопределение пометок для этих двух столбцов. Также скорректирован код на C, который обращался к `srsublsn`, не проверяя отличие этого значения от null; сбой в этом месте был маловероятен, но всё же не исключён.
- Исправление обработки ссылок `LATERAL` в условиях, связанных с неразворачиваемым вложенным `SELECT` в предложении `FROM` (Том Лейн)

  Следствием исправленного теперь упущения мог быть крах сервера или сбой проверочного утверждения во время выполнения соответствующего запроса.
- Отказ от предположения об отсутствии кортежей в сторонних таблицах, которые ещё не анализировались (Том Лейн)

  Ошибочное предположение, прежде всего, сказывалось на оценке планировщиком количества групп, которое должно быть получено с `GROUP BY`.
- Удаление ненужного предупреждения об «оставшемся кортеже-местозаполнителе» при сбросе обобщения индекса BRIN (Альваро Эррера)

  Эта ситуация вполне возможна и ожидаема после отмены очистки, поэтому данное предупреждение было скорее лишним шумом.
- Улучшение обработки ошибок в серверном модуле `buffile` (Томас Манро)

  Исправлено поведение в некоторых случаях, когда ошибки ввода/вывода не считались отличными от достижения конца файла или вообще игнорировались. Также в те сообщения, где это уместно, добавлены детали: номера блоков и количества байтов.
- Устранение аномалий при проверке конфликтов в режиме изоляции `SERIALIZABLE` (Питер Гейган)

  Когда добавляемый параллельно кортеж изменяется ещё одной параллельной транзакцией и ни одна версия кортежа не видна в снимке текущей транзакции, при проверке конфликта сериализации могло возникнуть неверное представление о том, как этот кортеж соотносится

с результатами текущей транзакции. В итоге сериализуемая транзакция могла успешно зафиксироваться, тогда как она должна отменяться с ошибкой сериализации.

- Предотвращение многократной пометки «мёртвых» элементов в индексе btree, уже имеющих такую пометку (Масахико Савада)

  Хотя это никак не вредило функциональности, в режиме включённых контрольных сумм или при включении `wal_log_hints` в журнал WAL вносились ненужные записи.

- Окружение блокировками операций с файлом `pg_control` в тех местах кода, где таких блокировок не хватало (Натан Боссарт, Фудзии Масао)

  В результате упущения файл `pg_control` мог быть перезаписан с неверной контрольной суммой, что могло создать проблемы впоследствии, вплоть до невозможности запустить базу данных в случае сбоя сервера до следующего изменения `pg_control`.

- Исправление ошибок в `currtid()` и `currtid2()` (Микаэль Пакье)

  В этих функциях (недокументированных и используемых только старыми версиями драйвера ODBC) нашлись дефекты, которые могли приводить к краху сервера или к странным сообщениям вида «could not open file» (не удалось открыть файл) при попытке использования этих функций с отношениями, не хранящимися на диске.

- Очистка кода от недопустимых конструкций, в которых `elog()` или `palloc()` вызываются при установленной циклической блокировке (Микаэль Пакье, Том Лейн)

  В логике, связанной со слотами репликации, обнаружилось несколько подобных конструкций. Хотя вероятность проявления проблемы мала, ошибка в вызванной функции могла приводить к зависанию блокировки.

- Исправление проверочного утверждения в подписчике логической репликации, не позволявшего использовать `REPLICA IDENTITY FULL` (Эйлер Тавейра)

  Ошибка была исключительно в проверочном утверждении, так что в обычных выпускаемых сборках она никак не проявлялась.

- Исправление сообщений об ошибках при нехватке места, выдаваемых программами pg_dump и pg_basebackup (Джастин Призби, Том Лейн, Альваро Эррера)

  В некоторых местах кода выдавались нелепые сообщения вида «could not write file: Success» (не удалось записать файл: Успех).

- Исправление параллельного восстановления таблиц, для которых заданы права доступа и на уровне таблицы, и на уровне столбцов (Том Лейн)

  Назначения прав на уровне таблицы должны применяться в первую очередь, но при параллельном восстановлении порядок назначения прав не гарантировался, и поэтому могли возникать ошибки «tuple concurrently updated» (кортеж изменён параллельно) или исчезали некоторые назначения прав на уровне столбцов. Это исправление заключается в добавлении зависимостей между такими элементами в файле архива; это означает, что для предотвращения проблемы необходимо сделать новую копию базы с применением исправленного pg_dump.

- Установка при выполнении pg_upgrade нулевого значения `vacuum_defer_cleanup_age` в целевом кластере (Брюс Момджян)

  Если конфигурация целевого кластера была изменена и параметру `vacuum_defer_cleanup_age` было присвоено ненулевое значение, это мешало корректно заморозить кортежи в системных каталогах, вследствие чего обновление могло прерваться странным образом. Теперь гарантируется, что данный параметр получает подходящее значение на время обновления.

- Добавление в pg_recvlogical цикла чтения всех ожидающих обработки сообщений перед штатным завершением (Ной Миш)

Если эти сообщения не получить, передатчик данных репликации может обнаружить сбой при передаче и завершиться, не установив нужную позицию LSN для слота репликации. Вследствие этого, данные повторно передавались при следующем подключении. Также могли быть пропущены сообщения об ошибках, переданные после последних данных, подлежащих обработке программой pg_recvlogical.

- Исправление в pg_rewind реакции на исчезновение файлов в исходном каталоге данных (Джастин Призби, Микаэль Пакье)

  Когда в качестве исходной используется работающая база данных, удаления файлов возможны в любой момент времени, но программа pg_rewind приходила в замешательство, вызывая `stat()` для файла, который исчезал после того, как был просканирован содержащий его каталог.

- Переключение pg_test_fsync в двоичный режим ввода/вывода в Windows (Микаэль Пакье)

  Ранее эта утилита записывала тестовый файл в текстовом режиме, что не соответствует фактическому поведению PostgreSQL.

- Исправление дефекта при инициализации локального состояния в `contrib/dblink` (Джо Конвей)

  При определённых обстоятельствах он мог приводить к тому, что функция `dblink_close()`, вопреки ожиданиям, выполняла на удалённой стороне `COMMIT`.

- Исправление в `contrib/pgcrypto` ошибочного использования `deflate()` (Том Лейн)

  Функции `pgp_sym_encrypt` могли выдавать неправильные сжатые данные вследствие нарушения требований API zlib. Нам не сообщали о проявлениях этой ошибки со стандартной библиотекой zlib, но они определённо наблюдаются с библиотекой zlibNX, разработанной IBM.

- Устранение ошибки в алгоритме распаковки данных в функциях `pgp_sym_decrypt` модуля `contrib/pgcrypto`, проявлявшейся в особом случае (Кётаро Хоригути, Микаэль Пакье)

  Поток сжатых данных вполне может заканчиваться пустым пакетом, но функция распаковки не принимала это и выдавала сообщение об испорченных данных.

- Переход к использованию соответствующей стандарту POSIX функции `strsignal()` вместо `sys_siglist[]` из BSD (Том Лейн)

  Тем самым решена проблема сборки с самыми последними версиями glibc.

- Обеспечение поддержки нашего кода NLS с Microsoft Visual Studio 2015 или новее (Хуан Хосе Сантамария Флеча, Давиндер Сингх, Амит Капила)

- Предупреждение возможного сбоя в нашем установочном скрипте для MSVC в случае наличия файла `configure` на несколько уровней выше каталога с исходным кодом (Арнольд Мюллер)

  В таком случае логика поиска файла `configure` могла принять каталог с найденным выше файлом за верхний уровень дерева исходных кодов.

E.37. Выпуск 10.13

Дата выпуска: 2020-05-14

В этот выпуск вошли различные исправления, внесённые после версии 10.12. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.37.1. Миграция на версию 10.13

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Если вы обновляете сервер с более ранней версии, чем 10.11, см. также Раздел E.39.

E.37.2. Изменения

- Сохранение свойства `indisclustered` у индексов, перезаписываемых командой `ALTER TABLE` (Амит Ланготе, Джастин Призби)

  Ранее команда `ALTER TABLE` теряла информацию о том, для какого индекса выполнялась команда `CLUSTER`.

- Сохранение свойства идентификации реплики у индексов, перезаписываемых командой `ALTER TABLE` (Цюань Цзунлян, Питер Эйзентраут)

- Блокирование объектов на более ранней стадии выполнения `DROP OWNED BY` (Альваро Эррера)

  Это позволяет исключить ошибки в условиях гонки, когда какие-либо из удаляемых объектов параллельно удаляются в другом сеансе.

- Исправление обработки ошибок при выполнении `CREATE ROLE ... IN ROLE` (Эндрю Гирт)

  В некоторых случаях при ошибке вместо надлежащего сообщения выдавалось «unexpected node type» (неожиданный тип узла) или что-то подобное.

- Предоставление всем членам роли `pg_read_all_stats` доступа ко всем статистическим представлениям, согласно ожиданиям (Магнус Хагандер)

  Нижележащие функции представлений `pg_stat_progress_*` ранее не считали эту роль имеющей соответствующие полномочия.

- Исправление в полнотекстовом поиске обработки отрицания для результата поиска фразы (Том Лейн)

  Запросы вида `!(foo<->bar)` не могли найти нужные строки, когда применялся поиск по индексу GiST или GIN.

- Исправление полнотекстового поиска в ситуациях, когда в поисковой фразе оказывается элемент, задающий и искомый префикс, и вес (Том Лейн)

- Улучшение выбора выдержек из документа в функции `ts_headline()` при обработке фразовых запросов (Том Лейн)

- Устранение ошибок в обработке `gin_fuzzy_search_limit` (Аде Хэйуорд, Том Лейн)

  При небольшом значении `gin_fuzzy_search_limit` выполнение запросов могло неожиданно замедляться из-за непреднамеренного многократного сканирования одной и той же страницы индекса. В другом месте кода желаемая фильтрация не применялась вовсе, в результате чего могло возвращаться слишком много значений.

- Добавление в набор форматов, допустимых для типа `circle`, формата «`(`*x*`,`*y*`),`*r*», который должен приниматься согласно документации (Дэвид Чжан)

- Доработка функций `get_bit()` и `set_bit()` для поддержки строк `bytea` длиннее 256 МБ (Мувад Ли)

  Так как аргумент, задающий номер бита, имеет тип `int4`, с помощью этих функций нельзя обращаться к битам за пределами первых 256 мегабайт значения `bytea`. В 13 версии тип аргумента будет расширен до `int8`, а тем временем в текущей версии обеспечена поддержка этими функциями больших значений `bytea` в обозначенном пределе.

- Игнорирование ошибок «файл не найден» в `pg_ls_waldir()` и родственных функциях (Том Лейн)

  Это устраняет сбой в условиях гонки, когда файл удаляется между моментом, когда мы получили запись о нём в каталоге, и моментом, когда мы пытаемся получить дополнительную информацию о нём, вызывая `stat()`.

- Устранение возможной утечки открытого дескриптора каталога в `pg_ls_dir()`, `pg_timezone_names()`, `pg_tablespace_databases()` и родственных функциях (Джастин Призби)
- Исправление разрешения типов в полиморфных функциях, чтобы фактический тип результата `anyarray` выводился из входного типа, если на вход поступает только `anyrange` (Том Лейн)
- Устранение возможности сбоя при прерывании команды `REINDEX` сигналом завершения сеанса (Том Лейн)
- Устранение маловероятного сбоя в случае нарушения условий ограничений в секционированных таблицах (Андрес Фройнд)
- Предотвращение отображения мусорных данных при выводе статистики таблицы соединения по хешу в `EXPLAIN` (Константин Книжник, Том Лейн, Томас Манро)
- Исправление расчёта длительности этапов в процедуре усечения кучи при выполнении `VACUUM VERBOSE` (Тацухито Касахара)
- Обеспечение передачи состояний ожидания TimelineHistoryRead и TimelineHistoryWrite во всех местах в коде, где читаются и записываются файлы истории линий времени (Масахиро Икеда)
- Предотвращение повторного добавления «waiting» (ожидание) в заголовок процесса, выводимый PS (Масахико Савада)
- Предотвращение преждевременного перерабатывания сегментов WAL во время восстановления после сбоя (Жеан-Гийом де Рорте)

  Сегменты WAL, которые становились готовыми к архивации в ходе процедуры восстановления, могли перерабатываться, не попадая в архив.
- Отказ от сканирования неактуальных линий времени в ходе восстановления архива (Кётаро Хоригути)

  Теперь не будут производиться многократные попытки считать несуществующие файлы WAL из архивного хранилища, что важно при низкоскоростном подключении к архиву.
- Удаление ненужной проверки, сигнализирующей об ошибке «subtransaction logged without previous top-level txn record» (записи подтранзакции в журнале не предшествует запись транзакции верхнего уровня) (Арсений Шер, Амит Капила)

  Ситуация, которая ранее считалась недопустимой, могла сложиться на вполне законных основаниях, поэтому данную проверку следует удалить.
- Обеспечение снятия блокировки `io_in_progress_lock` для слота репликации при обработке нештатных ситуаций (Паван Деоласи)

  Ранее блокировка снималась не всегда, вследствие чего процесс walsender мог зависнуть в ожидании.
- Устранение условий гонки при управлении синхронными ведомыми серверами (Том Лейн)

  В процессе применения изменений параметра `synchronous_standby_names` существовало окно, в котором могли быть приняты неверные решения об освобождении транзакций, ожидающих синхронной фиксации. Подобные неверные решения могли приниматься и тогда, когда после завершения процесса walsender он сразу же заменялся другим.
- Недопущение изменения `nextXid` в обратном направлении на ведомом сервере (Ека Паламадай)

  В результате условия гонки сервер горячего резерва мог передавать ведущему серверу некорректные ответные сообщения, вследствие чего на ведущем операция `VACUUM` могла выполняться раньше времени.
- Добавление значений SQLSTATE в формируемые отчёты об ошибках в нескольких местах (Масахико Савада)

- Реализация в PL/pgSQL надёжного метода пресечения попытки выполнить функцию событийного триггера как обычную функцию (Том Лейн)
- Устранение утечки памяти в libpq при использовании режима `sslmode=verify-full` (Роман Пешкуров)

  При проверке сертификата на стадии установления соединения была возможна утечка памяти. Это могло стать серьёзной проблемой, если клиентский процесс в своём жизненном цикле устанавливал много подключений к базе данных.
- Обработка в ecpg аргумента «-» как означающего «читать с stdin» на всех платформах (Том Лейн)
- Добавление дополнения табуляцией имени файла для команды psql `\gx` (Вик Фиринг)
- Добавление в pg_dump поддержки конструкции `ALTER ... DEPENDS ON EXTENSION` (Альваро Эррера)

  Ранее программа pg_dump игнорировала зависимости, добавленные таким образом, вследствие чего они терялись после выгрузки/восстановления или выполнения pg_upgrade.
- Исправление в pg_dump выгрузки комментариев к объектам политики RLS (Том Лейн)
- Перенос в выгрузке pg_dump восстановления событийных триггеров на финальный этап (Фабрицио де Ройес Мелло, Хамид Ахтар, Том Лейн)

  Это снижает риск нежелательного влияния событийных триггеров на восстановление других объектов.
- Исправление обработки кавычек в значениях параметров `--encoding`, `--lc-ctype` и `--lc-collate` утилиты createdb (Микаэль Пакье)
- Устранение краха при попытке вызвать функцию `lo_manage()` из `contrib/lo` как обычную функцию, а не в качестве триггера (Том Лейн)
- Защита от переполнения полей длины в типах `ltree` и `lquery` модуля `contrib/ltree` (Никита Глухов)
- Устранение утечки ссылки на кеш в `contrib/sepgsql` (Майкл Ло)
- Исправление обработки имён локалей в стиле Unix на платформе Windows (Хуан Хосе Сантамария Флеча)
- Использование pkg-config для определения расположения libxml2 в процедуре configure (Хью Макмастер, Том Лейн, Питер Эйзентраут)

  Если утилита pkg-config отсутствует или не имеет информации о libxml2, мы будем использовать xml2-config, как и прежде.

  Это изменение может нарушить процедуры сборки PostgreSQL, в которых для использования нестандартной версии libxml2 путь к нужной версии xml2-config добавляется в `PATH`. Теперь вместо этого нужно указать путь к нужному xml2-config в переменной окружения `XML2_CONFIG`. Этот метод будет работать и с новыми, и со старыми версиями PostgreSQL.
- Включение `CFLAGS_SL` в `CXXFLAGS` при сборке разделяемой библиотеки (Алексей Клюкин)

  Тем самым обеспечивается корректная компиляция исходных файлов C++, например при необходимости может добавляться `-fPIC`.
- Исправление в сборках MSVC манипуляции с путём к Python с учётом того, что он может содержать пробелы (Виктор Вагнер)
- Исправление в сборках MSVC определения версии Visual Studio во избежание зависимости от языковых настроек (Эндрю Дунстан)
- Использование в сборках MSVC ключа `-Wno-deprecated` с bison версии новее 3.0, как уже делается в сборках для отличных от Windows ОС (Эндрю Дунстан)

- Обновление данных часовых поясов до версии tzdata 2020a, включающее изменение правил перехода на летнее время в Марокко и канадском Юконе, а также корректировку исторических данных для Шанхая.

  Часовой пояс America/Godthab был переименован в America/Nuuk в соответствии с принятым сейчас названием; старое обозначение сохранено для совместимости в виде ссылки.

  Также был актуализирован список известных initdb часовых поясов в Windows, так что теперь параметры системного часового пояса на этой платформе будут восприняты корректно с большей вероятностью.

E.38. Выпуск 10.12

Дата выпуска: 2020-02-13

В этот выпуск вошли различные исправления, внесённые после версии 10.11. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.38.1. Миграция на версию 10.12

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Если вы обновляете сервер с более ранней версии, чем 10.11, см. также Раздел E.39.

E.38.2. Изменения

- Добавление отсутствовавших проверок прав для `ALTER ... DEPENDS ON EXTENSION` (Альваро Эррера)

  При обозначении объекта зависимым от расширения никакие проверки прав доступа не выполнялись. В результате этого упущения любой пользователь мог сделать так, чтобы процедуры, триггеры, матпредставления или индексы можно было удалить, имея право удалить какое-либо расширение. Теперь требуется, чтобы выполняющий эту команду пользователь был владельцем целевого объекта (это значит, что он сам может удалить этот объект). (CVE-2020-1720)

- Исправление кода подписчика логической репликации, чтобы должны образом вызывались триггеры `UPDATE`, связанные со столбцами (Питер Эйзентраут)

- Устранение сбоя при логическом декодировании в случаях, когда большую транзакцию приходится сохранять во множестве отдельных временных файлов (Амит Хандекар)

- Устранение угрозы сбоев или повреждения данных при обработке операции изменения строки подписчиком логической репликации (Том Лейн, Томаш Вондра)

  Обнаруженный дефект мог проявляться, только если в таблице подписчика имелись столбцы, не копируемые с публикующего сервера, и использовались типы данных, передаваемые по ссылке.

- Устранение в коде подписчика логической репликации сбоя, возникавшего после изменений структуры отношений, входящих в подписку (Жеан-Гийом де Рорте, Вигнеш Си)

- Ликвидация сбоя, возникавшего в подписчике логической репликации после отказа и перезапуска базы данных (Вигнеш Си)

- Увеличение эффективности логической репликации с применением `REPLICA IDENTITY FULL` (Константин Книжник)

  Когда выполняется поиск существующего кортежа в процессе операции изменения или удаления, должен возвращаться первый подходящий кортеж, а не последний.

- Предотвращение преждевременного отключения узлов плана Gather или GatherMerge, находящихся под узлом Limit (Амит Капила)

Тем самым устраняется возможность ошибки в случае, когда этот узел плана нужно сканировать многократно, например, если он оказался внутри вложенного цикла.

- Устранение утечки памяти, возникавшей при нехватке свободных слотов динамической общей памяти (Томас Манро)
- Аннулирование указания `CONCURRENTLY` для операций создания, удаления или перестроения индекса во временных таблицах (Микаэль Пакье, Хейкки Линнакангас, Андрес Фройнд)

  Тем самым исключаются странные ошибки, возникавшие с временными таблицами, для которых назначалось действие `ON COMMIT`. В любом случае, использовать `CONCURRENTLY` для временных таблиц (а это влечёт значительные издержки) не имеет смысла, так как никакие другие сеансы к таким таблицам обращаться не могут.
- Устранение ошибки, возникавшей при сбросе индексов по выражениям во временных таблицах с характеристикой `ON COMMIT DELETE ROWS` (Том Лейн)
- Ликвидация возможности сбоя при операциях с индексом BRIN, в котором используются типы данных `box`, `range` и `inet` (Хейкки Линнакангас)
- Исправление обработки удалённых страниц в индексах GIN (Александр Коротков)

  В результате устранены риски возникновения взаимоблокировок, некорректного обновления состояния удалённой страницы, а также риск ошибки при обходе недавно удалённой страницы.
- Предотвращение сбоя, возможного при выполнении вложенного `SELECT` (узла SubPlan) в списке `VALUES`, выдающем несколько строк (Том Лейн)
- Устранения сбоя, возможного после ошибки в FileClose() (Ной Миш)

  Эту проблему можно было наблюдать, только если был включён параметр `data_sync_retry`, так как в противном случае ошибка в FileClose() приводит к аварийной остановке сервера (PANIC).
- Предупреждение маловероятного риска сбоя при передаче переходного состояния агрегата по ссылке (Андрес Фройнд, Фёдор Сигаев)
- Улучшение вывода ошибок в функциях `to_date()` и `to_timestamp()` (Том Лейн, Альваро Эррера)

  Сообщения о неправильном названии месяца или дня во входных строках могли обрезаться в середине многобайтного символа, в результате чего сообщение могло быть неприемлемым для выбранной кодировки, что вызывало ошибки при дальнейшей его обработке. Теперь эти сообщения будут обрезаться по следующему пробельному символу.
- Устранение ошибочного сдвига на 1 результата функции `EXTRACT(ISOYEAR FROM` *`timestamp`*`)` с датами до нашей эры (Том Лейн)
- Недопущение переполнения стека при обращении к представлениям `information_schema` в случае существования в системных каталогах представлений, которые ссылаются на себя же (Том Лейн)

  Представление, ссылающееся на себя, работать не будет, так как при обращении к нему возникнет бесконечная рекурсия. Мы обрабатывали это корректно при попытке чтения из такого представления, но не учли, что зацикливание возможно и при проверке представлений на предмет поддержки автообновления.
- Вывод NULL в качестве времени начала транзакции для процессов walsender в `pg_stat_activity` (Альваро Эррера)

  Ранее в столбце `xact_start` иногда могло отображаться время запуска процесса.
- Увеличение производительности соединений по хешу с очень большими внутренними отношениями (Томас Манро)

- Предотвращение сбоев в особых случаях и исправление неверных оценок при вычислении избирательности для операторов диапазона `<@` и `@>` (Микаэль Пакье, Андрей Бородин, Том Лейн)
- Исключение из рассмотрения системных столбцов при обращении к расширенной статистике по самым частым значениям (Томаш Вондра)

  Тем самым предотвращаются ошибки планировщика «negative bitmapset member not allowed» (отрицательный элемент в битовой карте не допускается) при выполнении запросов с системными столбцами.
- Исправление логики индексов BRIN для поддержки гипотетических индексов BRIN (Жюльен Руо, Хейкки Линнакангас)

  Ранее, если расширение, «помогающее выбрать индекс», подталкивало планировщик к плану с гипотетическим индексом BRIN, это приводило к ошибке, так как код оценки стоимости BRIN всегда пытался физически прочитать метастраницу индекса. Теперь добавлена проверка, является ли индекс гипотетическим; и в этом случае считается, что индекс имеет параметры по умолчанию.
- Улучшено информирование об ошибках при попытках использовать автоматическое обновление представлений с правилами `INSTEAD` по условию (Дин Рашид)

  Это не поддерживалось и раньше, но ошибка проявлялась только во время выполнения запроса, так что её могли скрыть другие ошибки планировщика.
- Недопущение включения составного типа в себя же косвенным образом через диапазонный тип (Том Лейн, Жюльен Руо)
- Запрет на использование в ключе разбиения выражений, возвращающих псевдотипы, такие как `record` (Том Лейн)
- Корректировка сообщения об ошибке при попытке создать индекс по выражению с недопустимыми типами (Амит Ланготе)
- Исправление выгрузки представлений, содержащих только список `VALUES`, после переименования выходного столбца представления (Том Лейн)
- Рассмотрение типов данных и правил сортировки в конструкциях `XMLTABLE` при вычислении зависимостей представления или правила (Том Лейн)

  Ранее было возможно нарушить работу представления с `XMLTABLE`, удалив тип, который не фигурировал нигде больше в представлении. Данное исправление не корректирует зависимости ранее существовавших представлений, но повлияет на зависимости будущих.
- Предотвращение нежелательного понижения регистра символов и сокращения параметров аутентификации RADIUS (Маркос Давид)

  Ранее при разборе `pg_hba.conf` эти поля воспринимались как идентификаторы SQL, но это некорректно в общем случае.
- Исправление поведения `NOTIFY`, чтобы поступающие уведомления передавались клиенту до `ReadyForQuery`, а не после (Том Лейн)

  В результате гарантируется, что с libpq и другими клиентскими библиотеками, работающими по тому же алгоритму, клиент сможет получить все уведомления к моменту, когда транзакция будет считаться завершённой. Возможно, это не имеет большого значения для реальных приложений (так как им в любом случае придётся иметь дело с асинхронными уведомлениями), но это упрощает создание тестов с воспроизводимым поведением.
- Реализация в libpq возможности разобрать все связанные с GSS параметры подключения даже при компиляции кода без поддержки GSSAPI (Том Лейн)

  Таким образом в этом аспекте устранено отличие от поддержки SSL, в отношении которой давно было принято решение, что имеет смысл всегда принимать все параметры, даже если

они игнорируются или не могут работать в связи с отсутствием некоторой функциональности в конкретной сборке.

- Исправление некорректной обработки кодов формата `%b` и `%B` в функции ecpg `PGTYPEStimestamp_fmt_asc()` (Томаш Вондра)

  В результате ошибки со сдвигом на один с этими кодами могло выводиться неправильное название месяца либо мог произойти сбой.

- Исправление поведения pg_dump/pg_restore в параллельном режиме в случае ошибки создания рабочих процессов (Том Лейн)
- Предотвращение сбоя или блокировки при попытке завершить выполнение pg_dump/pg_restore сигналом (Том Лейн)
- Рассмотрение подтипов массивов и диапазонов в поиске типов данных, не поддерживающих обновление, в pg_upgrade (Том Лейн)
- Исправление проверки синтаксиса параметра createuser `--connection-limit` (Альваро Эррера)
- Устранение сбоя в `postgres_fdw` при попытке передать удалённому серверу команду вида `UPDATE remote_tab SET (x,y) = (SELECT ...)` (Том Лейн)
- Ограничение в `contrib/dict_int` минимально допустимого значения `maxlen` числом 1 (Томаш Вондра)

  Тем самым предотвращаются сбои с бессмысленными значениями данного параметра.

- Введение в реализованной в `contrib/tablefunc` функции `crosstab()` запрета на значения NULL в категориях (Джо Конвей)

  Такой вариант использования никогда не давал разумных результатов, а на некоторых платформах мог приводить к сбою.

- Добавление пометки `PGDLLIMPORT` для некоторых переменных GUC, связанных с тайм-аутами и статистикой, чтобы их могли использовать расширения в Windows (Паскаль Легран)

  Это изменение затронуло переменные `idle_in_transaction_session_timeout`, `lock_timeout`, `statement_timeout`, `track_activities`, `track_counts` и `track_functions`.

- Устранение утечки памяти в коде проверки целостности контекстов памяти «slab» (Томаш Вондра)

  Это не является проблемой в производственных сборках, так как в них проверка контекстов памяти обычно отключена; однако утечка могла быть весьма заметной в отладочных сборках.

- Устранение дублирования статистических записей, выдаваемых механизмом сбора статистики LWLock (Фудзии Масао)

  Код статистики LWLock (он не собирается по умолчанию, но может быть собран при компиляции с ключом `-DLWLOCK_STATS`) мог в результате некорректного формирования ключа хеш-таблицы выдавать несколько записей для одного сочетания блокировки и обслуживающего процесса.

- Устранение условий гонки, при которых могла задерживаться доставка межпроцессорных сигналов в Windows (Амит Капила)

  В частности, это проявлялось в странных нарушениях синхронизации сообщений `NOTIFY`.

- Добавление нескольких повторений для операции открытия файла в случае получения ошибки `ERROR_ACCESS_DENIED` в Windows (Александр Лахин, Том Лейн)

  Это помогает справиться с ситуацией, когда попытка открыть файл не удаётся из-за того, что файл помечен для удаления, но ещё не удалён окончательно. Например, утилита pg_ctl

часто сталкивалась с такой ошибкой, пытаясь определить, остановился ли главный процесс (postmaster).

E.39. Выпуск 10.11

Дата выпуска: 2019-11-14

В этот выпуск вошли различные исправления, внесённые после версии 10.10. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.39.1. Миграция на версию 10.11

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Однако если вы используете расширение `contrib/intarray` с индексом GiST и применяете оператор `<@` для поиска по индексу, примите к сведению соответствующее замечание в списке изменений.

Если вы обновляете сервер с более ранней версии, чем 10.6, см. также Раздел E.44.

E.39.2. Изменения

- Устранение сбоя команды `ALTER TABLE SET` с нестандартным параметром отношения (Микаэль Пакье)
- Недопущение изменения типа столбца, унаследованного от нескольких таблиц, в случае, когда изменяются не все родительские таблицы (Том Лейн)

  Ранее это допускалось, что приводило к сбоям в запросах к потерявшим согласованность родительским таблицам.
- Исключение попыток заморозить в ходе `VACUUM` идентификатор старой мультитранзакции, включающей всё ещё выполняющуюся транзакцию (Натан Боссарт, Джереми Шнайдер)

  В подобной ситуации выполнение `VACUUM` прерывалась ошибкой, пока не завершалась старая транзакция.
- Исправление в планировщике проверки, возможна ли смена регистра для символа, выполняемой в `ILIKE` при использовании правил сортировки ICU (Том Лейн)

  В результате ошибки планировщик считал фиксированным префиксом слишком большую часть шаблона поиска `ILIKE`, вследствие чего при поиске по индексу могли не находиться записи, удовлетворяющие шаблону.
- Реализация должной обработки выражений смещения в предложениях `WINDOW` при манипуляциях с выражениями запросов (Эндрю Гирт)

  В результате устранённого теперь упущения использование нетривиальных выражений в качестве смещений могло вызывать разнообразные ошибки. Например, в случае встраивания функции обращение к её параметрам в таком выражении могло производиться некорректно.
- Исправление обработки переменных-кортежей в выражениях `WITH CHECK OPTION` и выражениях политик защиты на уровне строк (Андрес Фройнд)

  Ранее с подобными конструкциями могли возникать ошибки с некорректными сообщениями о несовпадении типа табличной строки.
- Устранение сбоя главного процесса при попытке выделить для параллельного выполнения запроса фоновый исполнитель в момент отсутствия свободных слотов в массиве дочерних процессов (Том Лейн)
- Предотвращение двойного освобождения памяти в случае, когда триггер `BEFORE UPDATE` возвращает старый кортеж «как есть» и такой триггер не последний (Томас Манро)

- Передача ожидаемого контекста ошибки в случае возникновения ошибки при изменении параметров GUC в ходе запуска параллельного исполнителя (Томас Манро)
- Обеспечение установления в сериализуемом режиме предикатных блокировок на уровне строки в корректной версии строки (Томас Манро, Хейкки Линнакангас)

  При изменении видимой строки по методу HOT блокировка могла устанавливаться в предыдущей и ставшей неактуальной версии, что приводило к нарушению гарантии сериализации.
- Осуществление вызова `fsync()` только для тех файлов, которые открываются для чтения/записи (Андрес Фройнд, Микаэль Пакье)

  В некоторых местах кода были попытки вызывать эту функцию и для файлов, открываемых только для чтения, но на некоторых платформах при этом возникают ошибки вида «bad file descriptor» (плохой дескриптор файла).
- Возможность успешного преобразования кодировки для более длинных строк, чем раньше (Альваро Эррера, Том Лейн)

  Ранее существовало жёсткое ограничение размера входной строки в 0.25 ГБ, но сейчас преобразование будет успешным, если результат преобразования умещается в 1 ГБ.
- Исключение ненужного поиска в каталоге при зачистке страницы кучи (Томас Манро)

  Теперь в этом месте нет необходимости в проверке наличия нежурналируемых индексов, которая вызывала снижение производительности при определённой нагрузке. К тому же эта проверка теоретически могла быть причиной взаимоблокировок.
- Выбор по возможности более компактных хранилищ кортежей для оконных функций (Эндрю Гирт)

  В некоторых случаях в хранилище кортежей могли попадать все столбцы исходных таблиц, а не только те, что требуются для запроса.
- Возможность освобождения памяти функцией `repalloc()` при значительном уменьшении размера большого блока (Том Лейн)
- Обеспечение удаления временных файлов WAL и истории по завершении восстановления архива (Савада Масахико)
- Устранение ошибки, не позволяющей восстановить архив при включении параметра `recovery_min_apply_delay` (Масао Фудзии)

  Параметр `recovery_min_apply_delay` в такой конфигурации обычно не используется, тем не менее он должен работать.
- Устранение ошибки при логической репликации, когда публикующий сервер и подписчик имеют разные представления о столбцах идентификации реплики (Жеан-Гийом де Рорте, Питер Эйзентраут)

  Если на подписчике частью ключа идентификации реплики объявлялся столбец, вовсе отсутствующий на публикующем сервере, при репликации выдавались ошибки «negative bitmapset member not allowed» (отрицательный элемент в битовой карте не допускается).
- Недопущение нежелательной задержки при отключении процесса walsender, участвующего в логической репликации (Крейг Рингер, Альваро Эррера)
- Исправление обработки тайм-аута в процессах walreceiver, участвующих в логической репликации (Жюльен Руо)

  Ошибочная логика препятствовала использованию `wal_receiver_timeout` при логической репликации.
- Исправление передачи меток времени в сообщениях репликации для логического декодирования (Джефф Джейнс)

В частности, в результате допущенной ошибки поле `pg_stat_subscription.last_msg_send_time` обычно содержало NULL.

- Исправление обработки подтранзакций при восстановлении снимка в процессе логического декодирования (Масахико Савада)

  Ликвидированная теперь ошибка проявлялась в сбоях проверочных утверждений; в обычных сборках её последствия неизвестны.

- Устранение условий гонки при завершении обслуживающего процесса, который до этого оказывался в состоянии ожидания синхронной репликации (Донмин Лю)

- Исправление поведения `ALTER SYSTEM` при дублировании переменных в `postgresql.auto.conf` (Иэн Барвик)

  Сама команда `ALTER SYSTEM` не может привести файл `postgresql.auto.conf` в такое состояние, но это могут сделать внешние программы. Теперь дублирующиеся назначения переменных будут удаляться, а новое значение (если оно назначается) будет добавляться в конце файла.

- Недопущение в конфигурационных файлах указаний включения каталогов с пустыми именами и более явное информирование о рекурсивном включении файлов (Иэн Барвик, Том Лейн)

- Ликвидация сообщений о брошенных соединениях при использовании аутентификации PAM (Том Лейн)

  Клиенты на базе libpq обычно делают две попытки подключения, когда требуется пароль, так как они спрашивают пароль только в случае неудачи при первой попытке. Поэтому на стороне сервера приняты меры, чтобы в журнал не добавлялись бесполезные сообщения о том, что клиент закрыл подключение при запросе пароля. Однако в коде проверки PAM об этом забыли, и сервер выдавал сообщения о фантомных ошибках при проверке подлинности.

- Исправление разбора входной строки `time with time zone`, содержащей неполную дату (Александр Лахин)

  В случае указания часового пояса, смещение которого от UTC меняется во времени, также должна указываться дата, чтобы это смещение можно было определить. В некоторых случаях, в зависимости от используемого синтаксиса, эта проверка могла не работать, и выдавались невразумительные результаты.

- Исправление некорректного поведения `bitshiftright()` (Том Лейн)

  Оператор сдвига битовой строки вправо не заполнял корректно нулями биты в последнем байте результата при длине битовой строки не кратной 8. Хотя эти биты не видны для большинства операций, их содержимое могло влиять на результат сравнения, так как при сравнении битовых строк дополнительные биты специально не обрабатываются (предполагается, что они всегда нулевые).

  Если в ваших таблицах могли сохраниться некорректные данные в результате выполнения `bitshiftright()`, их можно исправить примерно так:

  ```
  UPDATE mytab SET bitcol = ~(~bitcol) WHERE bitcol != ~(~bitcol);
  ```

- Устранение сбоя при выборе узла пространства имён в `XMLTABLE` (Чепмен Флэк)

- Исправление проверки целочисленного переполнения при умножении интервалов в особых случаях (Юя Ватари)

- Ликвидация утечек памяти в функциях `lower()`, `upper()` и `initcap()` при использовании правил сортировки ICU (Константин Книжник)

- Устранение сбоев при обработке некорректных данных аффиксов в словарях текстового поиска `ispell` (Артур Закиров)

- Исправление некорректной логики сжатия для списков идентификаторов GIN (Хейкки Линнакангас)

  Для одного элемента в списке идентификаторов может требоваться 7 байт, если расстояние между соседними идентификаторами проиндексированных кортежей превышает 16 ТБ. В одном месте логика обработки этого списка не учитывала это, и значение могло записываться в 6-байтный буфер. В принципе это могло приводить и к выходу за границы стека, но на большинстве архитектур следующий байт оказывался в блоке выравнивания, так что это не причиняло вреда. В любом случае вызвать эту ошибку было очень сложно.

- Исправление обработки значений infinity, NaN и NULL в KNN-GiST (Александр Коротков)

  Порядок результатов запроса мог быть некорректным (отличаться от порядка простой сортировки результатов), если для отличных от NULL значений столбцов получались расстояния infinity или NaN.

- Исправление поиска NULL в KNN-SP-GiST (Никита Глухов)

- Поддержка ещё одного варианта названия локали норвежского букмола «Norwegian (Bokmål)» в Windows (Том Лейн)

- Устранение ошибки компиляции клиентского кода ECPG, включающего заголовочный файл `ecpglib.h`, при определённом макросе `ENABLE_NLS` (Том Лейн)

  Этот дефект стал следствием некорректного размещения объявления функции `ecpg_gettext()` — она не должна быть видна в клиентском коде.

- Обновление внутренней информации о сервере в psql после неожиданной потери соединения и успешного переподключения (Петер Биллен, Том Лейн)

  Обычно в этом нет необходимости, так как информация о сервере будет той же, но в особых случаях подключение может устанавливаться к одному из нескольких серверов, и тогда это имеет смысл. В результате этого изменения psql теперь повторно выдаёт интерактивные сообщения, которые он раньше выдавал только при запуске, например, сообщает об использовании SSL.

- Ликвидация в коде psql обращения по нулевому указателю, которое наблюдалось на некоторых платформах (Квентин Рамо)

- Обеспечение ожидаемого порядка в выводе pg_dump одинаково названных триггеров и объектов политики защиты на уровне строк (Бенджи Гиллам)

  Ранее, если два триггера разных таблиц назывались одинаково, они сортировались по OID, тогда как предпочтительнее была бы сортировка по имени таблицы. То же наблюдалось и с политиками RLS.

- Восстановление совместимости pg_dump с серверами старее версии 8.3 (Том Лейн)

  После одного из предыдущих исправлений pg_dump всегда пытался обратиться к каталогу `pg_opfamily`, который появился только в версии 8.3.

- Реализация в pg_restore прочтения указания `-f -` как «выводить в stdout» (Альваро Эррера)

  Тем самым поведение pg_restore синхронизируется с поведением других приложений, и в частности, в версиях до 12 это теперь работает так же, как и в pg_restore в 12-ой версии, что упрощает создание скриптов выгрузки/восстановления, работающих с разными версиями PostgreSQL. До этого изменения программа pg_restore интерпретировала это указание как «выводить в файл с именем -», но такое поведение мало кому нужно.

- Улучшение в pg_upgrade проверок использования типа данных, представление которого изменилось, например `line` (Томаш Вондра)

  Предыдущая реализация работала некорректно, когда такой тип данных оказывался базовым для домена или составного типа.

- Выявление ошибок чтения файлов в процессе работы pg_basebackup (Дживан Чок)
- Устранение в pg_basebackup излишних вызовов fsync() для выходных файлов; выполнение этого вызова только в конце резервного копирования (Микаэль Пакье)

  Ранее в случае медленной синхронизации файлов были возможны сбои из-за тайм-аутов.
- Отключение тайм-аутов в pg_rewind при подключении к работающему кластеру, по подобию pg_dump (Александр Кукушкин)
- Исправление ошибки при работе pg_waldump с ключом `-s`, когда запись продолжения WAL заканчивается ровно на границе страницы (Андрей Лепихов)
- Включение в pg_waldump поля `newitemoff` в записи разделения страниц btree (Питер Гейган)
- Устранение ненужного перевода строк в выводе pg_waldump с ключом `--bkp-details` при описании WAL-записей, отражающих запись полных страниц (Андрес Фройнд)
- Устранение небольшой утечки памяти в pg_waldump (Андрес Фройнд)
- Улучшение поведения vacuumdb с большим значением `--jobs` в случае нехватки файловых дескрипторов (Michael Paquier)
- Игнорирование в `contrib/amcheck` нежурналируемых индексов при работе в режиме горячего резерва (Андрей Бородин, Питер Гейган)

  В этом контексте нежурналируемые индексы не обязательно должны быть в актуальном состоянии, поэтому проверять их не нужно.
- Исправление в `contrib/intarray` классов операторов GiST для корректной работы оператора `<@` с пустыми массивами (Том Лейн)

  Предложение вида *`столбец_массив <@ константный_массив`* считается индексируемым, но при поиске по индексу могут не находится пустые массивы, тогда как они тривиальным образом удовлетворяют этому условию.

  Единственным вариантом исправления, подходящим для переноса в старые версии, видится требование сканирования всего индекса для оператора `<@`, что и было реализовано. К сожалению, это означает, что такой поиск будет менее эффективен, чем простое последовательное сканирование.

  Для приложений, производительность которых значительно пострадает от этого изменения, можно предложить два выхода: переключиться на индексы GIN, в которых нет такого дефекта, или заменить конструкцию *`столбец_массив <@ константный_массив`* на *`столбец_массив <@ константный_массив`* `AND` *`столбец_массив && константный_массив`*. Таким образом будет получена примерно та же производительность, что и раньше, и будут найдены все непустые подмножества данного константного массива, чего и можно было ранее ожидать от изначального запроса.
- Исправление в configure проверки наличия libperl для совместимости с последними выпусками Red Hat (Том Лейн)

  Ранее проверка могла не работать, когда пользователь устанавливал в `CFLAGS` значение `-O0`.
- Обеспечение корректной генерации кода циклических блокировок (spinlock) на PowerPC (Ной Миш)

  В предыдущей реализации таких блокировок компилятор мог выбрать нулевой регистр для использования в ассемблерной инструкции, которая этот регистр не принимает. Следствием этого были ошибки при сборке. Мы обнаружили глубоко в истории только одно сообщение о подобной ошибке, но вообще проблемы могли возникать у тех, кто собирал модифицированный код PostgreSQL или использовал нестандартные параметры компилятора.
- Ликвидация зависимости от функции компилятора xlc, `__fetch_and_add()` на PowerPC (Ной Миш)

Компилятор xlc версии 13 и новее интерпретирует эту функцию не так, как мы рассчитываем, в результате чего сборка PostgreSQL оказывается нерабочей. Для исправления был написан специальный ассемблерный код.

- Отказ от использования ключа компилятора `-qsrcmsg` в AIX (Ной Миш)

  Таким образом удалось обойти внутреннюю ошибку компилятора xlc v16.1.0, и это повлияло лишь на формат выдаваемых компилятором сообщений об ошибках.

- Исправление процедуры сборки MSVC для корректной обработки пробелов в пути к OpenSSL (Эндрю Дунстан)

- Обновление данных часовых поясов до версии tzdata 2019c, включающее изменения правил перевода на летнее время на Фиджи и острове Норфолк, а также корректировку исторических данных для Альберты, Австрии, Бельгии, Британской Колумбии, Камбоджи, Гонконга, штата Индиана (графства Перри), Калининграда, штатов Кентукки и Мичиган, острова Норфолк, Южной Кореи и Турции.

E.40. Выпуск 10.10

Дата выпуска: 2019-08-08

В этот выпуск вошли различные исправления, внесённые после версии 10.9. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.40.1. Миграция на версию 10.10

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Если вы обновляете сервер с более ранней версии, чем 10.6, см. также Раздел E.44.

E.40.2. Изменения

- Требование указания схемы при приведении к временному типу с использованием синтаксиса вызова функции (Ной Миш)

  Мы давно требуем, чтобы вызов временных функций записывался с явным указанием временной схемы, примерно так: `pg_temp.`*`имя_функции`*`(`*`аргументы`*`)`. Теперь её необходимо указывать и в приведении к временным типам с использованием функциональной записи: `pg_temp.`*`имя_типа`*`(`*`аргумент`*`)`. В противном случае имеется возможность перехватить вызов функции, используя временный объект, что позволит повысить привилегии примерно тем же способом, который мы блокировали в CVE-2007-2138. (CVE-2019-10208)

- Устранение сбоя в `ALTER TABLE ... ALTER COLUMN TYPE` при изменении типов нескольких столбцов в одной команде (Том Лейн)

  Тем самым устранена регрессия, которая появилась в предыдущих корректирующих выпусках: индексы, построенные по модифицируемым столбцам, обрабатывались неправильно, что приводило к странным ошибкам в процессе `ALTER TABLE`.

- Корректировка зависимостей с целью недопущения удаления столбцов в ключе секционирования (Том Лейн)

  Команда `ALTER TABLE ... DROP COLUMN` не позволяет непосредственно удалить столбец, входящий в ключ разбиения. Однако при косвенном удалении (например, при каскадном удалении, начиная с типа данных этого столбца) необходимая проверка не выполнялась, что позволяло удалить ключевой столбец. В результате секционированная таблица приходила в негодность — обратиться к ней было нельзя и удалить её тоже.

  В результате этого исправления в `pg_depend` добавляются записи, гарантирующие, что при каскадном удалении ключевого столбца будет удалён не только он, но и вся

секционированная таблица. Заметьте, что такие записи будут добавляться только при создании секционированной таблицы; то есть данное исправление не устраняет риск повреждения уже существующих секционированных таблиц. Однако проблема может возникнуть только со столбцами в ключе разбиения, которые имеют пользовательские типы данных, поэтому это не должно представлять угрозы для большинства пользователей.

- Сохранение в `GROUP BY` значимости столбцов таблицы-родителя в иерархии наследования (Дэвид Роули)

  Обычно когда в предложении `GROUP BY` фигурируют столбцы первичного ключа таблицы, все остальные группирующие столбцы можно считать незначащими, так как первых вполне достаточно для получения уникальных групп. Но такая оптимизация некорректна, если запрос также обращается к дочерним таблицам — уникальность в родительской таблице не распространяется на дочерние.

- Устранение ненужных этапов сортировки в некоторых запросах с предложением `GROUPING SETS` (Эндрю Гирт, Ричард Гуо)

- Устранение сбоя при обращении к переходным таблицам триггера в процессе перепроверок `EvalPlanQual` (Александр Акципетров)

  Триггеры, работающие с переходными таблицами, иногда могли ломаться при одновременном изменении таблиц.

- Исправление поведения внешних ключей, содержащих несколько столбцов, при перестроении ограничения внешнего ключа (Том Лейн)

  Команда `ALTER TABLE` могла неправильно определять, нужно ли перепроверять внешний ключ, если столбцы в ключе имели разные типы. По всей видимости проявлялось это всегда консервативным образом, то есть перепроверка производилась и тогда, когда это не требовалось.

- Отключение сбора расширенной статистики в деревьях наследования (Томаш Вондра)

  Это устраняет ошибку с сообщением «tuple already updated by self» (кортеж уже изменился сам) в процессе `ANALYZE`.

- Устранение неоправданных взаимоблокировок при повышении уровня блокировки кортежа (Алексей Клюкин)

  Когда две или более транзакций ждут, пока транзакция T1 освободит блокировку на уровне кортежа, а T1 при этом повышает уровень блокировки, по завершении T1 между этими ожидающими транзакциями может возникнуть неоправданная взаимоблокировка.

- Устранение проблемы при разрешении взаимоблокировок, в которых задействовано несколько параллельных исполнителей (Жуй Хай Цзян)

  Маловероятно, что эту ошибку можно было получить не синтетическими запросами, но в этом исключительном случае запросы, попавшие в ситуацию взаимоблокировки (в идеале разрешаемую автоматически), потребовалось бы отменить вручную.

- Исправление преобразования диапазонов дат с бесконечными границами `infinity` с приведением их к канонической форме (Лауренц Альбе)

  Если граница диапазона — бесконечность, пытаться преобразовать открытый диапазон в закрытый и наоборот путём увеличения или уменьшения граничного значения нельзя. Поэтому в подобных случаях диапазон не должен изменяться.

- Ликвидирована потеря дробных цифр при приведении очень больших значений типа `money` к типу `numeric` (Том Лейн)

- Исправление ассемблерного кода циклических блокировок для процессоров MIPS, чтобы он корректно работал на MIPS r6 (ЮньЦян Су)

- Игнорирование символа перевода каретки (`\r`) при разборе файлов описаний служб в libpq (Том Лейн, Микаэль Пакье)

  В некоторых случаях такие файлы с переводами строк в стиле Windows могли разбираться неправильно, что проявлялось в ошибках при подключении.

- Исключение некорректных вариантов дополнения табуляцией, предлагаемых после `SET` *`переменная`* `=` в psql (Том Лейн)
- Устранение небольшой утечки памяти в psql при выполнении команды `\d` (Том Лейн)
- Исправление в pg_dump порядка вывода пользовательских классов операторов (Том Лейн)

  Если пользовательский класс операторов оказывался классом операторов подтипа для пользовательского диапазонного типа, связанные объекты выводились в неправильном порядке; в результате выгруженные данные нельзя было восстановить. (Нижележащая ошибка в обработке зависимости классов операторов могла проявляться и в других случаях, но зафиксирован только этот.)

- Устранение возможности зависания pgbench при использовании ключа `-R` (Фабьен Коэльо)
- Обеспечение в `contrib/passwordcheck` возможности сосуществования с другими пользователями функции `check_password_hook` (Микаэль Пакье)
- Исправление проверок `contrib/sepgsql` для поддержки последних версий SELinux (Майк Пальмиотто)
- Улучшение стабильности регрессионных тестов `src/test/recovery` (Микаэль Пакье)
- Уменьшение объёма вывода в stderr в тестировочном скрипте pg_upgrade (Том Лейн)
- Исправление TAP-тестов для совместимости с Perl из среды msys в случаях, когда каталог сборки находится не внутри точки монтирования корня msys (Ной Миш)
- Обеспечение поддержки сборки Postgres с использованием Microsoft Visual Studio 2019 (Харибабу Комми)
- При сборке проектов Visual Studio переменная окружения `WindowsSDKVersion` не должна игнорироваться, если она установлена (Пэйфэн Цю)

  Это исправление решает проблемы сборки в некоторых конфигурациях.

- Поддержка в проектах Visual Studio библиотеки OpenSSL версии 1.1.0 и выше (Хуан Хосе Сантамария Флеча, Микаэль Пакье)
- Возможность передачи параметров make нижележащей утилите gmake при вызове на верхнем уровне make не из комплекта GNU (Томас Манро)
- Недопущение выбора вариантов `localtime` и `posixrules` в качестве значения `TimeZone` в процессе initdb (Том Лейн)

  В некоторых случаях программа initdb могла выбирать одно из таких имён несуществующих часовых поясов вместо имени «настоящего» часового пояса. Этим двум именам будет предпочитаться любое другое, представляющее часовой пояс на уровне C.

- Изменение представления `pg_timezone_names`: часовой пояс `Factory` будет отображаться, только если он имеет короткую аббревиатуру (Том Лейн)

  В прошлом администрация IANA ввела этот искусственный пояс с «аббревиатурой» `Local time zone must be set--see zic manual page` (Должен быть задан местный часовой пояс -- см. страницу руководства zic). Современные версии базы данных tzdb содержат для него аббревиатуру `-00`, но на ряде платформ эта база модифицируется так, что из неё продолжают исходить подобные старинные написания. Поэтому решено показывать этот часовой пояс, только если его аббревиатура действительно кратка.

- Синхронизация нашей копии библиотеки timezone с выпущенной IANA версией 2019b (Том Лейн)

В результате была добавлена поддержка нового ключа `-b slim` программы zic, что позволяет уменьшить размер устанавливаемых файлов часовых поясов. Эта возможность может пригодиться нам в будущем.

- Обновление данных часовых поясов до версии tzdata 2019b, включающее изменение правил перехода на летнее время в Бразилии, а также корректировку исторических данных для Гонконга, Италии и Палестины.

E.41. Выпуск 10.9

Дата выпуска: 2019-06-20

В этот выпуск вошли различные исправления, внесённые после версии 10.8. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.41.1. Миграция на версию 10.9

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Если вы обновляете сервер с более ранней версии, чем 10.6, см. также Раздел E.44.

E.41.2. Изменения

- Устранение возможности переполнения буфера при разборе верификатора SCRAM (Джонатан Кац, Хейкки Линнакангас, Микаэль Пакье)

  Любой прошедший проверку пользователь мог осуществить переполнение буфера в стеке, попытавшись сменить свой пароль на специально сконструированную строку. Помимо провоцирования сбоя PostgreSQL, этого могло быть достаточно для выполнения произвольного кода от имени пользователя ОС, запускающего PostgreSQL.

  Подобная возможность переполнения существовала в libpq, и эксплуатируя её, подставной сервер мог вызвать сбой клиентского приложения или, возможно, выполнить произвольный код от имени пользователя, запускавшего это приложение.

  Проект PostgreSQL благодарит Александра Лахина за сообщение об этой проблеме. (CVE-2019-10164)

- Устранение сбоя команды `ALTER TABLE ... ALTER COLUMN TYPE` при наличии в таблице частичного ограничения-исключения (Том Лейн)

- Ликвидация ошибки команды `COMMENT` при работе с комментариями ограничений доменов (Даниэль Густафссон, Микаэль Пакье)

- Предотвращение возможного повреждения памяти в случае повторения столбцов в списке ключей хеша при агрегировании по хешу (Эндрю Гирт)

- Исправление ошибочного построения планов с узлами MergeAppend (Том Лейн)

  Следствием этого дефекта могли быть сообщения об ошибках «could not find pathkey item to sort» (не удалось найти элемент ключа для сортировки).

- Исправление ошибочного вывода запросов с повторяющимися именами соединений (Филипп Дюбе)

  Результатом исправленного упущения могли быть ошибки при выгрузке/восстановлении представлений, образованных такими запросами.

- Исправление преобразования строковых значений JSON в выходные столбцы типа JSON в функциях `json_to_record()` и `json_populate_record()` (Том Лейн)

  В таких случаях должна выдаваться константа в виде отдельного значения JSON, но код работал неправильно, если строковое значение содержало символы, требующие экранирования.

- Исправление некорректной оптимизации квантификаторов `{1,1}` в регулярных выражениях (Том Лейн)

  Такие квантификаторы считались бесполезными и оптимизировались. Хотя в документации говорится, что они задают «жадное» поведение (или «нежадное» в случае «нежадной» вариации `{1,1}?`) для подвыражения, к которому они относятся, фактически они не действовали. Ошибка могла проявляться только с подвыражениями, содержащими группирующие скобки или обратные ссылки.

- Устранение возможности ошибок при инициализации данных `pg_stat_activity` для нового процесса (Том Лейн)

  Ранее были возможны сбои определённых операций, выполняемых внутри критической секции, например, преобразования строк, извлечённых из сертификата SSL, в кодировку базы данных. В подобных случаях могло произойти глобальное зависание базы данных из-за нарушения протокола обращения к общим данным `pg_stat_activity`.

- Устранение условий гонки при проверке, продолжает ли использоваться конфликтующим главным процессом ранее созданный сегмент общей памяти (Том Лейн)

- Исправление небезопасной реализации обработчика сигналов WAL-приёмника (Том Лейн)

  Таким образом полностью исключены маловероятные случаи, когда процесс приёмника WAL завершался аварийно или зависал, получив команду на выключение.

- Недопущение обращений к базам данных для проверки параметров в процессах, не подключённых к определённой базе данных (Вигнеш Си, Андрес Фройнд)

  Некорректные обращения могли вызывать ошибки вида «cannot read pg_class without having selected a database» (невозможно прочитать pg_class, не выбрав базу данных).

- Предупреждение возможного зависания в libpq при использовании SSL, когда объём данных в буфере очереди OpenSSL оказывается кратным 256 байтам (Дэвид Биндерман)

- Улучшение обработки в initdb нескольких равнозначных названий часового пояса системы (Том Лейн, Эндрю Гирт)

  Теперь initdb будет анализировать символическую ссылку `/etc/localtime`, если она существует, чтобы выбрать из всех равнозначных названий часового пояса системы наиболее подходящее. Таким образом при наличии нескольких названий часового пояса initdb скорее всего выберет именно то название, которое ожидает пользователь. Если же `/etc/localtime` не является символической ссылкой на файл данных часового пояса и часовой пояс не задан переменной окружения `TZ`, поведение initdb останется прежним.

  Кроме того, теперь `UTC` предпочитается другим названиям этого часового пояса, когда ни `TZ`, ни `/etc/localtime` не предлагают другой вариант. Тем самым устранено неудобство, возникшее после произошедшего в tzdata 2019a изменения, сделавшего названия часового пояса `UCT` и `UTC` равнозначными: initdb выбирал обозначение `UCT`, что не соответствовало ожиданиям пользователей.

- Исправление порядка команд `GRANT`, выдаваемых программами pg_dump и pg_dumpall для баз данных и табличных пространств (Натан Боссарт, Микаэль Пакье)

  В случае каскадных назначений прав при восстановлении мог произойти сбой, так как команды `GRANT` выдавались не в том порядке, в котором должны были с учётом взаимозависимостей.

- Изменение варианта пересоздания секций в pg_dump — теперь будет выдаваться команда `CREATE TABLE` с последующей `ATTACH PARTITION` вместо использования указания `PARTITION OF` в команде создания таблицы (Альваро Эррера, Дэвид Роули)

  Таким образом устраняются проблемы, связанные с возможным изменением порядка столбцов в соответствии с родительской таблицей. Кроме того, секция теперь будет

восстановлена из копии (в виде независимой таблицы), даже если её родительская таблица не восстановилась; при выполнении `ATTACH` произойдёт ошибка, но её можно просто игнорировать.

- Устранение дезориентирующих сообщений об ошибках в reindexdb (Жюльен Руо)
- Возвращение корректного состояния vacuumdb в случае ошибки при выполнении параллельных заданий (Жюльен Руо)
- Исправление `contrib/auto_explain` для недопущения ошибок с параллельными запросами (Том Лейн)

  Ранее параллельный исполнитель мог попытаться передать на анализ свой запрос, тогда как родительский запрос в `auto_explain` не обрабатывался. Иногда это могло работать, хотя само по себе это неправильно, а в некоторых случаях выдавались ошибки вида «could not find key N in shm TOC» (не удалось найти ключ N в оглавлении общей памяти).

  Также исправлена ошибка со сдвигом на один, в результате которой не обязательно отслеживались все запросы, даже при доле выборки, равной 1.0.
- В `contrib/postgres_fdw` теперь учитываются возможные модификации данных, производимые локальными триггерами `BEFORE ROW UPDATE` (Сёхей Мотидзуки)

  Если триггер изменял столбец, который не затрагивался командой `UPDATE`, новое значение не передавалось на удалённый сервер.
- Недопущение ошибки в Windows в случаях, когда для базы данных выбрана кодировка SQL_ASCII, а мы пытаемся вывести в журнал строку с не ASCII-символами (Ной Миш)

  В коде предполагалось, что такие строки всегда представлены в UTF-8, и если оказывалось, что это не так, происходила ошибка. Теперь же в журнал просто передаются непреобразованные байты.
- Обеспечение совместимости заголовочных файлов PL/pgSQL с C++ (Георгий Тарасов)

E.42. Выпуск 10.8

Дата выпуска: 2019-05-09

В этот выпуск вошли различные исправления, внесённые после версии 10.7. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.42.1. Миграция на версию 10.8

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Если вы обновляете сервер с более ранней версии, чем 10.6, см. также Раздел E.44.

E.42.2. Изменения

- Устранение возможности обойти политики защиты на уровне строк (RLS), используя оценки избирательности (Дин Рашид)

  При вычислении оценок избирательности планировщик применяет пользовательские операторы к значениям, находящимся в `pg_statistic` (то есть к самым частым значениям). Таким образом, негерметичный оператор может раскрыть некоторые данные в столбце таблицы, даже если пользователь не имеет доступа к этому столбцу. В CVE-2017-7484 мы добавили ограничения для предотвращения этой утечки, но не учли особенности защиты на уровне строк. Пользователь, которому в SQL дано право чтения столбца, но политикой RLS запрещён доступ к некоторым строкам, тем не менее мог получить некоторые сведения об их содержимом через негерметичный оператор. Это исправление усиливает ограничения, разрешая применение негерметичных операторов к статистическим данным только при отсутствии политики RLS. (CVE-2019-10130)

- Предупреждение повреждения каталога при создании в транзакции, состоящей из одного оператора, и временной таблицы с `ON COMMIT DROP`, и столбца идентификации (Питер Эйзентраут)

  Возможность повреждения здесь была не замечена ранее ввиду бессмысленности такой транзакции, так как временная таблица в ней должна удаляться сразу после создания.

- Предотвращение краха сервера при выполнении перепроверки EPQ для результирующего отношения запроса с секционированием (Амит Ланготе)

  Он происходил, когда использовался уровень изоляции `READ COMMITTED` и в другом сеансе параллельно изменялись какие-либо целевые строки.

- Исправление поведения `UPDATE` и `DELETE` при работе с деревом наследования или секционированной таблицей, допускающими исключение всех таблиц (Амит Ланготе, Том Лейн)

  Ранее в подобных условиях такие запросы не выдавали корректный набор выходных столбцов с предложением `RETURNING`. Помимо этого, в этих запросах также не срабатывали никакие триггеры уровня операторов.

- Устранение выдаваемых при наличии публикации `FOR ALL TABLES` некорректных ошибок при внесении изменений во временные и нежурналируемые таблицы (Питер Эйзентраут)

  Такие таблицы в контексте публикаций должны были полностью исключаться из рассмотрения, но исключались не везде.

- Исправление обработки явных элементов `DEFAULT` в команде `INSERT ... VALUES` с несколькими строками `VALUES`, целевым отношением для которой является обновляемое представление (Амит Ланготе, Дин Рашид)

  Когда в обновляемом представлении для столбца не задано значение по умолчанию, но оно задано в нижележащей таблице, команда `INSERT ... VALUES` с одной строкой вставит в столбец это значение по умолчанию. Однако при выполнении этой команды с несколькими строками вместо этого значения всегда вставлялся NULL. Теперь это исправлено, и такая команда выполняется так же, как и команда с одной строкой.

- Исправление `CREATE VIEW`, позволяющее создавать представления с нулём столбцов (Ашутош Шарма)

  Мы должны разрешить это для согласованности с возможностью создавать таблицы с нулём столбцов. Так как таблицу можно преобразовать в представление, ранее такое представление можно было получить даже при существовавшем ограничении, что приводило к сбоям при выгрузке/загрузке базы данных.

- Добавление поддержки `CREATE TABLE IF NOT EXISTS ... AS EXECUTE ...` (Андреас Карлссон)

  Сочетание предложений `IF NOT EXISTS` и `EXECUTE` должно поддерживаться, но в грамматике отсутствовало соответствующее правило.

- Использование прав корректного пользователя при выполнении вложенных `SELECT` в выражениях политик защиты на уровне строк (Дин Рашид)

  Ранее при обращении через представление к таблице с политикой RLS такие проверки могли осуществляться от имени пользователя, выполняющего запрос к представлению, а не от имени владельца представления, как должно быть.

- Теперь при значении `xml` параметра `xmloption` XML-документы считаются допустимыми значениями типа `xml` согласно требованию, появившемуся в стандарте SQL:2006 (Чепмен Флэк)

  Ранее PostgreSQL следовал определению стандарта SQL:2003, в котором это не допускалось. Но это создавало серьёзную проблему при выгрузке/восстановлении данных: ни одно из

значений `xmloption` не позволяло принять все допустимые XML-данные. В связи с этим решено перейти к определению 2006.

Также модифицирована утилита pg_dump, чтобы при восстановлении данных выполнялась команда `SET xmloption = content`. Это гарантирует, что выгрузка/восстановление сработает, даже если в данных преобладает вариант `document`.

- Улучшение выполняемых при запуске сервера проверок на предмет использования ранее созданного сегмента общей памяти (Ной Миш)

  Теперь главный процесс (postmaster) будет более надёжно определять активные процессы, оставшиеся от предыдущего воплощения сервера, даже в случае удаления файла `postmaster.pid`.

- Исключение транзакций параллельных исполнителей из числа отдельных транзакций.

- Устранение несовместимости записей GIN-индекса в WAL (Александр Коротков)

  Исправление, вошедшее в февральский корректирующий выпуск, не учитывало все тонкости обратной совместимости. Ведущий сервер этого выпуска формировал в WAL такие записи об удалении страниц GIN, которые не мог корректно воспринять ведомый более старой версии.

- Устранение возможности сбоя при выполнении команды `SHOW` через соединение репликации (Микаэль Пакье)

- Устранение утечки памяти при пересоздании в кеше структур, связанных с секционированными отношениями (Амит Ланготе, Том Лейн)

- Допущение кодов ошибок `EINVAL` и `ENOSYS` в результатах функций `fsync` и `sync_file_range` там, где это приемлемо (Томас Манро, Джеймс Сьюэлл)

  Предыдущее изменение, приводящее к панической остановке при ошибках файловой синхронизации, оказалось излишне параноидальным в ряде случаев, где сбой предсказуем и по сути означает «операция не поддерживается».

- Вывод в представлении `pg_stat_activity` правильного имени отношения при вычислении сводных данных BRIN в процессе автоочистки (Альваро Эррера)

- Устранение ошибок планировщика «failed to build any *N*-way joins» (не удалось построить никакие *N*-сторонние соединения) с подзапросами LATERAL снаружи полных внешних соединений (Том Лейн)

- Исправлено некорректное планирование запросов, в которых возвращающая множество функция применяется к заведомо пустому отношению (Том Лейн, Жюльен Руо)

  В 10 версии следствием этого упущения могли быть только чуть менее эффективные планы, тогда как в 11 могли выдаваться ошибки «set-valued function called in context that cannot accept a set» (функция, возвращающая множество, вызвана в контексте, где ему нет места).

- Проверка прав корректного пользователя при применении правил, позволяющих негерметичным операторам обращаться к данным `pg_statistic` (Дин Рашид)

  Когда пользователь обращается к нижележащей таблице через представление, решение о возможности применения негерметичных операторов к статистическим данным таблицы должно приниматься с учётом прав не этого пользователя, а владельца представления. Таким образом, правила планировщика по определению видимости данных приводятся в соответствие с правилами исполнителя, что позволит избежать неоправданно невыгодных планов.

- Ускорение планирования запросов со множеством условий равенства и множеством возможно связанных с ними ограничений внешнего ключа (Дэвид Роули)

- Оптимизация неэффективных операций сложности O(N^2) при отмене транзакции, создавшей множество таблиц (Томаш Вондра)

- Предупреждение краха сервера в особых случаях при выделении динамической общей памяти (Томас Манро, Роберт Хаас)
- Устранение условий гонки при управлении динамической общей памятью (Томас Манро)

  В этих условиях могли возникать ошибки «dsa_area could not attach to segment» (не удалось подключить dsa_area к сегменту) или «cannot unpin a segment that is not pinned» (нельзя открепить сегмент, который не закреплён).
- Устранение условий гонки, при которых сервер горячего резерва мог не отключиться, получив запрос на постепенное отключение (Том Лейн)
- Устранение возможности краха при получении в параметрах `pg_identify_object_as_address()` неправильных данных (Альваро Эррера)
- Ликвидация возможности ошибок «could not access status of transaction» (не удалось получить статус транзакции) в `txid_status()` (Томас Манро)
- Улучшение правил проверки паролей, зашифрованных алгоритмами SCRAM-SHA-256 и MD5 (Джонатан С. Кац)

  Строка пароля с определёнными начальными символами могла по ошибке приниматься за строку, закодированную в формате SCRAM-SHA-256 или MD5. Такой пароль можно было установить, но нельзя было использовать.
- Исправление обработки значений `lc_time`, подразумевающих кодировку, отличающуюся от кодировки базы (Хуан Хосе Сантамария Флеча, Том Лейн)

  С локализованными названиями дней или месяцев, включающими не ASCII-символы, в таких локалях ранее выдавались неожиданные ошибки или некорректные строки.
- Исправление ошибочных проверок `operator_precedence_warning` с унарным оператором «минус» (Рикард Фалькеборн)
- Недопущение `NaN` в качестве значения серверных параметров, принимающих числа с плавающей точкой (Том Лейн)
- Реорганизация работы `REINDEX` для предотвращения сбоев проверочных утверждений при переиндексировании отдельных индексов `pg_class` (Андрес Фройнд, Том Лейн)
- Исправление проверочных утверждений в планировщике при обработке параметризованных фиктивных путей (Том Лейн)
- Добавление нужной функции проверки в результат, выдаваемый функцией `SnapBuildInitialSnapshot()` (Антонин Хоуска)

  Для кода ядра это не важно, но важно для некоторых расширений.
- Устранение периодических ошибок «could not reattach to shared memory» (не удалось переподключиться к общей памяти) при запуске сеансов в Windows (Ной Миш)

  Невыявленным ранее источником таких проблем было создание стеков потоков в пуле потоков по умолчанию. Теперь такие стеки размещаются в другой области памяти.
- Исправлена обработка ошибок при сканировании каталога в Windows (Константин Книжник)

  Такие ошибки, как отсутствие прав для чтения каталога, не обнаруживались и не выдавались корректно; вместо этого поведение кода было таким же, как и с пустым каталогом.
- Исправление грамматических дефектов в ecpg (Том Лейн)

  Из-за пропущенной точки с запятой конструкция `SET` *`переменная`* `= DEFAULT` (но не `SET` *`переменная`* `TO DEFAULT`) в программах ecpg обрабатывалась некорректно, и в результате получался синтаксически некорректный код, не воспринимаемый сервером. Кроме того, в командах `DROP TYPE` и `DROP DOMAIN`, в которых перечислялось несколько имён типов, фактически обрабатывалось только первое имя.

- Согласование синтаксиса `CREATE TABLE AS` в ecpg с принятым на стороне сервера (Дайсукэ Хигути)
- Ликвидация возможности переполнения буфера при обработке в ecpg имён включаемых файлов (Лю Хуайлин, Фей Ву)
- Недопущение краха в `contrib/postgres_fdw` при наличии в запросе, использующем удалённую группировку или агрегирование, элемента списка `SELECT`, не связанного с вложенным запросом, внешней ссылкой или символом параметра (Том Лейн)
- Предупреждение краха в `contrib/vacuumlo` при ошибке в `lo_unlink()` (Том Лейн)
- Синхронизация нашей копии библиотеки timezone с выпущенной IANA версией 2019a (Том Лейн)

  При этом устранён небольшой дефект в zic, в результате которого выдавались данные о некорректном переходе в 2440 г. для часового пояса `Africa/Casablanca`, и добавлена поддержка нового параметра zic `-r`.
- Обновление данных часовых поясов до версии tzdata 2019a, включающее изменение правил перехода на летнее время в Палестине и Метлакатле, а также корректировку исторических данных для Израиля.

  Часовой пояс `Etc/UCT` теперь является ссылкой на `Etc/UTC` (добавленной для обратной совместимости), а не отдельным поясом, генерирующем аббревиатуру `UCT`, которая теперь, как правило, просто опечатка. PostgreSQL по-прежнему принимает на вход такую аббревиатуру, но уже не выдаёт её.

E.43. Выпуск 10.7

Дата выпуска: 2019-02-14

В этот выпуск вошли различные исправления, внесённые после версии 10.6. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.43.1. Миграция на версию 10.7

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Если вы обновляете сервер с более ранней версии, чем 10.6, см. также Раздел E.44.

E.43.2. Изменения

- Во избежание повреждения данных вместо повторения вызова `fsync()` при ошибке теперь выполняется аварийный останов (Крейг Рингер, Томас Манро)

  В ряде популярных операционных систем ядро сбрасывает буферы данных, не имея возможности сохранить их на диске, и выдаёт ошибку в `fsync()`. При этом повторный вызов `fsync()` будет успешным, но на самом деле данные уже потеряны, так что продолжение работы сервера чревато повреждением базы. В случае же аварийного останова мы можем воспроизвести данные из WAL, где в такой ситуации может остаться единственная копия данных. Хотя это, определённо, неэффективное и не очень красивое поведение, других вариантов нет, а подобные ситуации, к счастью, крайне редки.

  Для управления этим поведением был добавлен новый параметр сервера data_sync_retry; если вы уверены, что ядро вашей ОС не теряет незаписанные буферы данных при описанном сценарии, вы можете задать для `data_sync_retry` значение `on` и восстановить прежнее поведение.
- Отныне в документацию каждой версии включаются только замечания к выпускам, относящимся к данной основной версии, а не к выпускам данной и всех предшествующих версий, как было раньше (Том Лейн)

Дублирование информации, присущее прежней схеме, вышло за допустимые рамки. Теперь мы не дублируем замечания к выпускам в каждой версии, но планируем поддерживать их полный архив на сайте нашего проекта.

- Обеспечение контроля ограничений `NOT NULL` в секциях секционированной таблицы (Альваро Эррера, Амит Ланготе)
- Использование безопасного уровня блокировки таблицы при отсоединении секции (Альваро Эррера)

  Ранее блокировка была недостаточно сильной и позволяла параллельно производить с таблицей DDL-операции, что могло привести к нежелательным результатам.
- Устранение проблем с применением `ON COMMIT DROP` и `ON COMMIT DELETE ROWS` к секционированным таблицам и таблицам с потомками в иерархии наследования (Микаэль Пакье)
- Недопущение выполнения `COPY FREEZE` с секционированными таблицами (Дэвид Роули)

  Принципиально данный режим должен поддерживаться, но реализация может потребовать слишком сложных изменений, переносить которые в ранние версии рискованно.
- Предупреждение возможных взаимоблокировок при попытке блокирования множества страниц в буфере (Нишант Фну)
- Предупреждение взаимоблокировки процесса очистки GIN с параллельными операциями добавления в индекс (Александр Коротков, Андрей Бородин, Питер Гейган)

  Вследствие этого изменения пришлось частично отказаться от оптимизации, добавленной в версии 10.0 с целью сократить число индексных страниц, блокируемых при удалении страницы списка идентификаторов GIN. Было обнаружено, что она приводит к взаимоблокировкам, так что мы отключили её на время разбирательства.
- Недопущение взаимоблокировки запросов на сервере горячего резерва с воспроизведением операций удаления страниц индекса GIN (Александр Коротков)
- Устранение возможности сбоев при логической репликации в случае использования индексов с выражениями или предикатами (Питер Эйзентраут)
- Исключение бесполезного и дорогостоящего логического декодирования данных TOAST при перезаписи таблицы (Томаш Вондра)
- Исправление логики остановки подмножества передатчиков WAL при включении синхронной репликации (Пол Гуо, Микаэль Пакье)
- Устранение возможности сохранения некорректного значения идентификатора реплики в записи WAL об удалении кортежа (Стас Кельвич)
- Предотвращение некорректного применения оптимизации с пропуском WAL во время выполнения `COPY` в представление или стороннюю таблицу (Амит Ланготе, Микаэль Пакье)
- Повышение приоритета файлов истории WAL по отношению к другим файлам данных WAL при выборе архиватором очередных файлов для архивации (Дэвид Стил)
- Устранение возможности сбоя в `UPDATE` с множественным `SET` и вложенным `SELECT` в качестве источника данных (Том Лейн)
- Устранение сбоя в случае, когда libxml2 возвращает null вместо сообщения об ошибке (Серхио Конде Гомес)
- Устранение ложных ошибок разбора, связанных с группировкой, которые были вызваны несогласованной обработкой присваиваний с правилами сортировки (Эндрю Гирт)

  В некоторых случаях выражения, которые должны были считаться совместными, не считались таковыми, если они включали операции с типами данных, поддерживающими правила сортировки.

- Вместо презумпции герметичности функции сравнения, которую вызывают функции `LEAST()` или `GREATEST()`, реализована соответствующая проверка (Том Лейн)

  Собственно утечку информации из функций сравнения btree обычно сложно организовать, но в принципе это возможно.

- Исправление некорректного планирования запросов с вложенными циклами над и под узлом Gather в плане (Том Лейн)

  Если на обоих уровнях вложенного цикла требовалось передать одну и ту же переменную в правую сторону, мог быть построен некорректный план.

- Исправлено некорректное планирование запросов, в которых подзапрос LATERAL должен выполняться при сканировании сторонней таблицы (Том Лейн)

- Исправлена ошибка при оценке стоимости соединения слиянием, проявляющаяся в особом случае (Том Лейн)

  Планировщик мог предпочесть соединение слиянием, когда диапазон внешнего ключа был намного меньше диапазона внутреннего ключа, даже при наличии множества повторяющихся ключей с внутренней стороны, а это плохой выбор.

- Предотвращение увеличения времени планирования порядка O(N^2) с запросами, содержащими тысячи индексируемых выражений (Том Лейн)

- Улучшение обработки параллельно изменяемых строк в ходе `ANALYZE` (Джефф Джейнс, Том Лейн)

  Ранее строки, удаляемые текущими транзакциями, исключались из выборки операции `ANALYZE`, но обнаружилось, что это нарушает согласованность в большей степени, чем их включение. Теперь выборка по сути соответствует снимку MVCC на момент запуска `ANALYZE`.

- Операция `TRUNCATE` теперь игнорирует дочерние таблицы, являющиеся временными таблицами других сеансов (Амит Ланготе, Микаэль Пакье)

  Таким образом поведение `TRUNCATE` теперь соответствует поведению других команд. Ранее в таких случаях обычно происходил сбой.

- Исправление обновления счётчиков статистики в ходе `TRUNCATE` для нужной таблицы (Том Лейн)

  Если с опустошаемой таблицей была связана таблица TOAST, по ошибке сбрасывались счётчики последней.

- Корректная обработка `ALTER TABLE ONLY ADD COLUMN IF NOT EXISTS` (Грег Старк)

- Поддержка команды `UNLISTEN` в режиме горячего резерва (Шэй Роджански)

  Эта команда не должна ничего делать, так как `LISTEN` не допускается в режиме горячего резерва, но возможность выполнения этой пустой операции упрощает логику сброса состояния для клиентов.

- Исправление зависимостей ролей в списках прав для схем и типов данных (Том Лейн)

  В некоторых случаях имелась возможность удалить роль, которой были назначены права. Эта проблема не проявлялась сразу, но при выгрузке/восстановлении или обновлении могли возникнуть ошибки, в частности выражающиеся в неудачных попытках назначить права ролям с числовыми именами.

- Недопущение использования временной схемы сеанса в двухфазных транзакциях (Микаэль Пакье)

  Обращение к временным таблицам в таких транзакциях было запрещено уже давно, однако оставалась возможность выполнять другие недопустимые операции с временными объектами.

- Обеспечение корректного сброса кеша отношений после добавления или удаления ограничений внешнего ключа (Альваро Эррера)

  В результате устранённого теперь упущения в существующих сеансах могло не действовать только что созданное ограничение или продолжало действовать удалённое.

- Обеспечение должного обновления кешей отношений после переименования ограничений (Амит Ланготе)
- Приложение дополнительных усилий для удаления потерянных временных таблиц в процессе автоочистки, а также при выполнении `DISCARD TEMP` (Альваро Эррера)

  В результате будут аккуратнее вычищаться следы деятельности прерванных сеансов.

- Исправление воспроизведения операций микроочистки индекса GiST, устраняющее возможность получения несогласованного состояния в параллельных запросах на серверах горячего резерва (Александр Коротков)
- Недопущение досрочной утилизации пустых страниц индекса GIN, чреватой ошибками при параллельном поиске по индексу (Андрей Бородин, Александр Коротков)
- Исправление проявляющихся в граничных случаях ошибок преобразования чисел с плавающей точкой в целые (Эндрю Гирт)

  Значения, ненамного превышающие максимально возможное целое, могли приниматься, а затем, в результате переполнения, превращаться в минимально возможное целое. Кроме того, могли ошибочно не допускаться значения, которые при округлении должны были преобразовываться в минимальное или максимальное допустимое целое.

- В запросе аутентификации PAM значение `PAM_RHOST` не должно задаваться для соединений, установленных через сокет Unix (Томас Манро)

  Ранее эта переменная принимала значение `[local]`, которое как минимум было бесполезным, так как в ней ожидается имя узла.

- Недопущение в `client_min_messages` более высоких уровней, чем `ERROR` (Джона Харрис, Том Лейн)

  Ранее в этой переменной можно было установить уровень `FATAL` или `PANIC`, в результате чего подавлялась передача клиенту обычных сообщений об ошибках. Однако это противоречит гарантиям, заложенным в протоколе обмена данными PostgreSQL, и некоторые клиенты могут этого не понять. В уже выпущенных версиях исправление заключается в неявной подмене таких уровней значением `ERROR`, а начиная с версии 12, они просто не будут приниматься.

- Переход в ecpglib к использованию функции `uselocale()` или `_configthreadlocale()` вместо `setlocale()` (Михаэль Мескес, Том Лейн)

  Так как функция `setlocale()` не является внутрипотоковой и может быть даже непотокобезопасной, предыдущая реализация вызывала проблемы в многопоточных приложениях на базе ecpg.

- Исправление ошибочных результатов при получении числовых данных через область дескриптора SQLDA ecpg (Дайсукэ Хигути)

  Ранее значения с ведущими нулями копировались некорректно.

- Исправление метакоманды psql `\g` *`назначение`* для операции `COPY TO STDOUT` (Даниэль Верите)

  Ранее параметр *`назначение`* игнорировался, и копируемые данные всегда попадали в текущий поток вывода.

- Исправление обработки специальных символов для вывода psql в формате LaTeX (Том Лейн)

  Ранее обратная косая черта и некоторые другие знаки препинания из ASCII выводились некорректно, следствием чего были ошибки синтаксиса в документе или отображение некорректных символов.

- Исправление обработки программой pg_dump материализованных представлений с неявными зависимостями от первичных ключей (Том Лейн)

  В результате этой ошибки могли неправильно помечаться записи таких представлений в архиве, что приводило к безобидным предупреждениям «archive items not in correct section order» (в последовательности элементов архива нарушен порядок разделов); и, что более значимо, могли некорректно работать варианты избирательного восстановления, например с указанием таких меток в `--section`.

- Устранение сбоя, имевшего место на некоторых платформах при обращении по нулевому указателю, когда pg_dump или pg_restore пытались вывести ошибку (Том Лейн)

- Игнорирование ошибок `SIGPIPE` в случаях, когда `COPY FROM PROGRAM` прекращает читать вывод программы досрочно (Том Лейн)

  Такие случаи невозможны на практике именно с `COPY`, но могут иметь место при использовании `contrib/file_fdw`.

- Исправление в `contrib/hstore` вычисления хеша для пустых значений `hstore`, созданных в версии 8.4 или более ранней (Эндрю Гирт)

  В предыдущей реализации хеш для этих значений вычислялся не так, как для пустых значений `hstore`, созданных более поздней версией, вследствие чего соединения или агрегирования по хешу могли выполняться неправильно. Поэтому имеет смысл переиндексировать все хеш-индексы, построенные по столбцам `hstore`, если таблицы могут содержать данные, которые были внесены ещё в версии 8.4 или раньше и с тех пор никогда не проходили через выгрузку/восстановление.

- Устранение сбоев и ускорение операций при обработке данных большого объёма операторами класса `gist__int_ops` модуля `contrib/intarray` (Эндрю Гирт)

- Установление в скрипте configure следующего порядка поиска `python`: собственно `python`, затем `python3`, и только потом `python2` (Питер Эйзентраут)

  Это позволяет конфигурировать PL/Python, не задавая явно переменную окружения `PYTHON` на тех платформах, где теперь отсутствует программа `python` без версии.

- Добавление поддержки новых переменных `PG_CFLAGS`, `PG_CXXFLAGS` и `PG_LDFLAGS` в Makefile для pgxs (Кристоф Берг)

  Это упрощает организацию нестандартных процессов сборки расширений.

- Исправление написанных на Perl скриптов сборки, чтобы они не полагались на присутствие элемента «`.`» в пути поиска, так как в последних версиях Perl он отсутствует (Эндрю Дунстан)

- Устранение проблем с разбором параметров командной строки в OpenBSD (Том Лейн)

- Перемещение вызова `set_rel_pathlist_hook`, чтобы расширения могли использовать его для передачи частичных путей при выполнении параллельных запросов (КайГай Кохэй)

  Ожидается, что это никак не повлияет на существующие сценарии использования.

- Переход в названиях функций поддержки red-black tree (красно-чёрного дерева) от префикса `rb` к `rbt` (Том Лейн)

  Это позволяет избежать конфликта имён с функциями Ruby, нарушающего работу PL/Ruby. Хочется надеяться, что в результате не пострадают другие расширения.

- Обновление данных часовых поясов до версии tzdata 2018i, включающее изменения правил перехода на летнее время в Казахстане, Метлакатле и на Сан-Томе и Принсипи. Часовой пояс Кызылорды в Казахстане разделился на два, при этом появился новый пояс Asia/Qostanay (Азия/Костанай), а в некоторых областях смещение UTC не изменилось. Также скорректированы исторические данные для Гонконга и многочисленных тихоокеанских островов.

E.44. Выпуск 10.6

Дата выпуска: 2018-11-08

В этот выпуск вошли различные исправления, внесённые после версии 10.5. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.44.1. Миграция на версию 10.6

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Однако если вы используете расширение `pg_stat_statements`, прочитайте запись о нём в списке изменений.

Если вы обновляете сервер с более ранней версии, чем 10.4, см. также Раздел E.46.

E.44.2. Изменения

- Корректное заключение в кавычки имён переходных таблиц в командах `CREATE TRIGGER ... REFERENCING`, которые выдаёт pg_dump (Том Лейн)

  Отсутствием кавычек мог воспользоваться непривилегированный пользователь с целью получения прав суперпользователя при последующем восстановлении выгруженных данных или выполнении pg_upgrade. (CVE-2018-16850)

- Устранение сбоев, которые могли возникать в особых случаях в семействе функций `has_`*`объект`*`_privilege()` (Том Лейн)

  Эти функции должны возвращать NULL, а не выдавать ошибку при получении неверного OID. Некоторые из этих функций делали это и прежде, но не все. Кроме того, в функции `has_column_privilege()` на некоторых платформах могли иметь место и другие сбои.

- Исправление функции `pg_get_partition_constraintdef()`, чтобы при получении неверного OID отношения не происходил сбой, а выдавался NULL (Том Лейн)

- Недопущение замедления порядка O(N^2) в функциях match/split с регулярными выражениями при обработке длинных строк (Эндрю Гирт)

- Исправление дефекта в обработке стандартных многосимвольных операторов, следующих непосредственно за комментарием или символами + и – (Эндрю Гирт)

  Следствием данного упущения могли быть ошибки при разборе или неправильное определение приоритета операторов.

- Недопущение замедления порядка O(N^3) при обработке лексическим анализатором длинных строк из символов + или – (Эндрю Гирт)

- Исправление некорректного выполнения подчинённых планов при сканировании внешнего запроса в обратном направлении (Эндрю Гирт)

- Исправление ошибочного поведения `UPDATE/DELETE ... WHERE CURRENT OF ...` после перемещения курсора (Том Лейн)

  Курсор, сканирующий несколько отношений (в частности, дерево наследования), мог работать неправильно, возвращаясь к предыдущему отношению.

- Исправление в функции `EvalPlanQual` обработки узлов InitPlan, которые выполняются по условию (Эндрю Гирт, Том Лейн)

  Вследствие ошибки реализации могли возникать сложновоспроизводимые сбои или выдаваться неправильные результаты при одновременных запросах на изменение, содержащих изолированные вложенные `SELECT` в конструкции `CASE`.

- Недопущение создания секции в триггере, присоединённом к родительской таблице (Амит Ланготе)

В идеале такое создание можно было бы разрешить, но в данный момент оно блокируется во избежание сбоев.

- Устранение проблем с применением `ON COMMIT DELETE ROWS` к секционированной временной таблице (Амит Ланготе)
- Исправление проверок классов символов для корректной поддержки в Windows символов Unicode выше U+FFFF (Том Лейн, Кэндзи Уно)

  Эта ошибка проявлялась в операциях полнотекстового поиска, а также в работе модулей `contrib/ltree` и `contrib/pg_trgm`.
- Недопущение передачи вложенных `SELECT` с оконными функциями, а также с указаниями `LIMIT` или `OFFSET`, параллельным исполнителям (Амит Капила)

  В таких случаях поведение могло быть несогласованным из-за того, что разные исполнители получали разные результаты вследствие вариативности в порядке строк.
- Смена владельца последовательности, принадлежащей сторонней таблице, при выполнении `ALTER OWNER` для этой таблицы (Питер Эйзентраут)

  Операция смены владельца должна распространяться и на такие последовательности, но раньше сторонние таблицы она не затрагивала.
- Обеспечение обработки сервером уже полученных прерываний `NOTIFY` и `SIGTERM` до начала ожидания данных от клиента (Джефф Джейнс, Том Лейн)
- Устранение ошибки, приводившей к выделению лишней памяти для строки результатов `array_out()` (Кэйити Хиробэ)
- Предотвращение утечки памяти на время выполнения запроса при работе с `XMLTABLE` (Эндрю Гирт)
- Ликвидация утечки памяти при сканировании индекса SP-GiST (Том Лейн)

  Сколько-нибудь значительное проявление этой утечки наблюдалось, только когда для ограничения-исключения, использующего SP-GiST, в индекс поступало много записей.
- Закрытие файла сопоставлений в `ApplyLogicalMappingFile()` после завершения его использования (Томаш Вондра)

  Ранее дескриптор файла терялся, что могло в итоге приводить к сбоям при логическом декодировании.
- Устранение ошибки логического декодирования в случаях, когда сопоставленная таблица каталога постоянно перезаписывается, например, в результате действия `VACUUM FULL` (Андрес Фройнд)
- Предотвращение запуска сервера со значением `wal_level`, недостаточно большим для поддержки существующего слота репликации (Андрес Фройнд)
- Предотвращение сбоя в случаях, когда служебная команда создаёт бесконечную рекурсию (Том Лейн)
- Устранение в процессе инициализации горячего резерва проблемы дублирующихся XID, которые образуются при выполнении двухфазных транзакций на ведущем сервере (Микаэль Пакье, Константин Книжник)
- Исправление поведения событийных триггеров при обработке вложенных команд `ALTER TABLE` (Микаэль Пакье, Альваро Эррера)
- Передача параллельным исполнителям времени начала оператора и транзакции от родительского процесса (Константин Книжник)

  Тем самым исправлено поведение таких функций, как `transaction_timestamp()`, выполняющихся в параллельном исполнителе.

- Обеспечение корректного выравнивания при передаче расширенных значений данных параллельным исполнителям, что позволяет избежать сбоев на платформах, требовательных к выравниванию (Том Лейн, Амит Капила)
- Исправление логики переработки файла WAL для правильного выполнения этой операции на ведомых серверах (Микаэль Пакье)

  В зависимости от значения параметра `archive_mode` ведомый сервер мог не удалить некоторые файлы WAL, подлежащие удалению.
- Исправление обработки времени фиксации транзакций в процессе восстановления (Масахико Савада, Микаэль Пакье)

  Если отслеживание времени фиксации не было включено постоянно, в процессе восстановления мог произойти сбой при попытке получить время фиксации транзакции, для которой оно не было записано.
- Использование случайной затравки для `random()` в initdb, а также в сервере, запускаемом в режиме начальной загрузки и в монопольном режиме (Ной Миш)

  Практическая польза этого изменения состоит прежде всего в разрешении проблемы, когда программа initdb не могла использовать общую память POSIX, считая её недоступной, из-за конфликтов имён, вызванных использованием одинаковой затравки.
- Предотвращение повреждения общей памяти в логике DSA (Томас Манро)
- Возможность прерывания операции выделения памяти в DSM (Крис Трэверс)
- Предупреждение сбоя в параллельном исполнителе при загрузке расширения, пытающегося обратиться к системным кешам в своей функции инициализации (Томас Манро)

  Мы не считаем это удачным приёмом программирования расширений, но до внедрения параллельных запросов это работало, а значит должно поддерживаться и теперь.
- Исправление поведения при динамическом включении параметра `full_page_writes` (Кётаро Хоригути)
- Недопущение возможного сбоя в результате двойного освобождения памяти при повторном сканировании SP-GiST (Эндрю Гирт)
- Предотвращение некорректной компоновки функций в src/port и src/common на платформах BSD, использующих формат ELF, а также в HP-UX и Solaris (Эндрю Гирт, Том Лейн)

  Разделяемые библиотеки, загружаемые в адресное пространство сервера, могли использовать серверные версии этих функций, а не собственные реализации, как должно быть. Так как в поведении этих двух наборов функций имеются различия, это приводило к проблемам.
- Предупреждение возможного переполнения буфера при воспроизведении операции распаковки страницы GIN из WAL (Александр Коротков, Шивасубраманьян Рамасубраманьян)
- Предотвращение переполнения метастраницы хеш-индекса в случаях, когда размер `BLCKSZ` меньше значения по умолчанию (Дилип Кумар)
- Добавление ранее упущенного пересчёта контрольной суммы страницы в хеш-индексах (Амит Капила)
- Выполнение операции fsync, ранее упущенной, для каталога слотов репликации (Константин Книжник, Микаэль Пакье)
- Устранение неожиданных тайм-аутов при использовании `wal_sender_timeout` на медленном сервере (Ной Миш)
- Исправление вычисления подходящей точки согласованности WAL в процессах горячего резерва (Александр Кукушкин, Микаэль Пакье)

  Выбор правильной точки позволяет предотвратить некорректное поведение сразу после того, как ведомый сервер достигает согласованного состояния базы при воспроизведении WAL.

- Корректная остановка фоновых рабочих процессов при получении главным процессом запроса на быстрое отключение до завершения запуска базы (Александр Кукушкин)
- Обновление карты свободного пространства при воспроизведении из WAL изменений флагов замороженных/полностью видимых страниц (Альваро Эррера)

  Ранее мы не заботились об этом, считая, что в карте свободного пространства в любом случае нет важной информации. Однако если она сильно устаревает, это может привести к значительному снижению производительности после повышения ведомого сервера до основного. Эта карта в конце концов будет исправлена в результате обновлений, но мы хотим иметь её в хорошем состоянии раньше, поэтому стоит приложить дополнительные усилия и поддерживать её актуальность при воспроизведении WAL.
- Предотвращение преждевременного освобождения ресурсов параллельных исполнителей по окончании запроса или при достижении предельного числа кортежей (Амит Капила)

  В этот момент завершение исполнителя допустимо, только если вызывающий процесс не может запросить обратное сканирование.
- Отказ от вызова обработчиков atexit при обработке `SIGQUIT` (Хейкки Линнакангас)
- Исключение сопоставлений пользователей для сторонних серверов из состава расширений (Том Лейн)

  При выполнении `CREATE USER MAPPING` в скрипте расширения для создаваемого сопоставления добавлялась зависимость, чего не должно быть. Сопоставления пользователей, как и роли, не могут быть членами расширений.
- Исправление поведения syslogger в случае ошибок при попытке открыть файлы CSV (Том Лейн)
- При передаче в libpq нескольких имён узлов они должны разрешаться в DNS не все сразу, а каждое в свою очередь (Том Лейн)

  Это предотвращает отказы, которых можно избежать, и замедления подключения после успешного соединения с одним из предыдущих серверов в списке.
- Исправление поведения libpq, чтобы тайм-ауты при подключении корректно применялись для отдельного имени или IP-адреса узла (Том Лейн)

  Ранее при некоторых условиях таймер при переключении на следующий узел не перезапускался, что могло приводить к преждевременному тайм-ауту.
- Исправление кода psql, а также примеров в документации, чтобы функция `PQconsumeInput()` вызывалась перед `PQnotifies()` (Том Лейн)

  Тем самым решена проблема, когда psql не выдавал полученное сообщение `NOTIFY` до следующей команды.
- Исправление в pg_dump обработки ключа `--no-publications`, чтобы с ним также игнорировались таблицы публикации (Жиль Даролд)
- Исключение из выгрузки pg_dump идентификационной последовательности при исключении её родительской таблицы (Дэвид Роули)
- Устранение возможной несогласованности при сортировке различных имён объектов в pg_dump (Джейкоб Чемпион)
- Исправление pg_restore, чтобы при выдаче команд `DISABLE`/`ENABLE TRIGGER` имя таблицы дополнялось схемой (Том Лейн)

  Это предотвращает сбои, возможные при новой политике выполнения восстановления с ограниченным путём поиска.
- В pg_upgrade исправлена обработка событийных триггеров в расширениях (Харибабу Комми)

Ранее pg_upgrade не сохранял связь событийных триггеров с расширениями.

- Исправление проверки состояния кластера в pg_upgrade, чтобы она корректно выполнялась на ведомом сервере (Брюс Момджян)
- Ограничение числа размерностей в значениях `cube` во всех функциях `contrib/cube` (Андрей Бородин)

  Ранее в некоторых функциях, связанных с кубами, можно было создать такие значения, которые затем не принимала функция `cube_in()`, что вызывало ошибки при восстановлении выгруженных данных.
- В `contrib/pg_stat_statements` роли `pg_read_all_stats` запрещено выполнение `pg_stat_statements_reset()` (Харибабу Комми)

  Роли `pg_read_all_stats` должно позволяться только чтение статистики, но не её изменение, поэтому разрешение на выполнение этой функции ей было дано некорректно.

  Чтобы это изменение вступило в силу, выполните `ALTER EXTENSION pg_stat_statements UPDATE` в каждой базе данных, где установлено расширение `pg_stat_statements`.
- Недопущение передачи расширением `contrib/postgres_fdw` предложений `ORDER BY`, не содержащих переменных, на удалённый сервер (Эндрю Гирт)
- Исправление функции `unaccent()` в расширении `contrib/unaccent`, чтобы она использовала словарь текстового поиска `unaccent`, находящийся в той же схеме, где и сама функция (Том Лейн)

  Ранее она пыталась найти словарь по пути поиска, но могла не обнаружить его, если путь поиска был ограничен.
- Устранение проблем при сборке в macOS 10.14 (Mojave) (Том Лейн)

  Усовершенствование скрипта configure, чтобы в `CPPFLAGS` добавлялся ключ `-isysroot`; без этого PL/Perl и PL/Tcl нельзя сконфигурировать или собрать в macOS 10.14. Значение sysroot можно переопределить во время конфигурирования или сборки, установив переменную `PG_SYSROOT` в аргументах configure или make.

  Теперь рекомендуется, чтобы для связанных с Perl расширений во флагах компилятора указывалось `$(perl_includespec)`, а не `-I$(perl_archlibexp)/CORE`. Второй вариант по-прежнему будет работать на большинстве платформ, но не в последних macOS.

  Также теперь не требуется указывать вручную ключ `--with-tclconfig`, чтобы собрать PL/Tcl в последних версиях macOS.
- Исправление скриптов сборки с MSVC и регрессионного тестирования для работы с последними версиями Perl (Эндрю Дунстан)

  Это изменение вызвано тем, что Perl теперь по умолчанию не включает текущий каталог в свой путь поиска.
- Реализована возможность запускать регрессионные тесты в Windows с учётной записью администратора (Эндрю Дунстан)

  Чтобы это было безопасно, pg_regress теперь лишает себя расширенных прав при запуске.
- Снятие запрета на возврат из функций сравнения btree значений `INT_MIN` (Том Лейн)

  До настоящего времени мы не позволяли типозависимым функциям сравнения возвращать `INT_MIN`, благодаря чему вызывающий код мог сменить порядок сортировки, просто изменив знак результата сравнения. Однако это могло вызывать проблемы с функциями сравнения, возвращающими непосредственно результат `memcmp()`, `strcmp()` и подобных функций, так как POSIX не накладывает на него никаких ограничений. Как минимум некоторые

последние версии `memcmp()` могут возвращать `INT_MIN`, что приводило к нарушению порядка сортировки. В связи с этим мы убрали это ограничение. Там же, где требуется поменять порядок сортировки на противоположный, теперь нужно использовать макрос `INVERT_COMPARE_RESULT()`.

- Устранение риска рекурсии при обработке сообщений об аннулировании общего кеша (Том Лейн)

  Эта ошибка могла привести, например, к сбою при обращении к системному каталогу или индексу, который был только что обработан командой `VACUUM FULL`.

  В результате этого изменения функция `LockAcquire` теперь может возвращать новый код результата, что теоретически может вызвать проблемы при использовании этой функции, хотя для этого схема её использования должна быть весьма необычной. Также был изменён API функции `LockAcquireExtended`.

- Сохранение и восстановление глобальных переменных SPI в `SPI_connect()` и `SPI_finish()` (Чепмен Флэк, Том Лейн)

  Это предотвращает накладки, возможные при вызове из одной функции, использующей SPI, другой такой функции.

- Предотвращение использования потенциально невыравненных буферов страниц (Том Лейн)

  Добавлены новые типы-объединения `PGAlignedBlock` и `PGAlignedXLogBlock`, которые должны использоваться вместо простых массивов char, чтобы компилятор гарантированно выравнивал адрес начала буфера. В результате устраняется риск аварийного сбоя на платформах, чувствительных к выравниванию, и может увеличиться быстродействие даже на тех платформах, где выравнивание не требуется.

- Приведение в `src/port/snprintf.c` значения результата `snprintf()` в соответствие со стандартом C99 (Том Лейн)

  На платформах, где этот код используется (в основном в Windows), его несовременное поведение (не соответствующие C99) могло препятствовать выявлению переполнения буфера, если вызывающий код рассчитывал на семантику C99.

- Требование обязательного использования `-msse2` при сборке компилятором clang для i386 (Андрес Фройнд)

  Тем самым решается проблема отсутствия проверок переполнения при операциях с плавающей точкой.

- Исправление в configure проверки, определяющей, результат какого типа возвращает `strerror_r()` (Том Лейн)

  Предыдущая реализация давала неправильный ответ при сборке с использованием компилятора icc в Linux (возможно, и в других случаях), в результате чего libpq не выдавала полезные сообщения при возникновении системных ошибок.

- Обновление данных часовых поясов до версии tzdata 2018g, включающее изменения правил перехода на летнее время в России (Волгограде), Чили, Марокко и на Фиджи, а также корректировку исторических данных для Китая, Гавайев, Японии, Макао и Северной Кореи.

E.45. Выпуск 10.5

Дата выпуска: 2018-08-09

В этот выпуск вошли различные исправления, внесённые после версии 10.4. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.45.1. Миграция на версию 10.5

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Если вы обновляете сервер с более ранней версии, чем 10.4, см. также Раздел E.46.

E.45.2. Изменения

- Осуществление полного сброса состояния libpq между попытками соединения (Том Лейн)

  Непривилегированный пользователь `dblink` или `postgres_fdw` мог обойти проверки, предназначенные для предотвращения использования учётных данных на стороне сервера, например, файла `~/.pgpass`, принадлежащего пользователю ОС, от имени которого работает сервер. В частности, уязвимость затрагивает серверы, принимающие локальные подключения с методом peer. Возможны были и другие атаки, например, SQL-инъекции в сеансе `postgres_fdw`. Для атаки на `postgres_fdw` через эту уязвимость пользователь должен был иметь возможность создавать объект стороннего сервера с нужными параметрами подключения, но атаку на `dblink` мог произвести любой пользователь, имеющий к нему доступ. Вообще любой пользователь, имеющий возможность менять параметры подключения для приложения на базе libpq, мог производить непредусмотренные действия, но другие эффективные сценарии эксплуатации этого сложно придумать. Мы благодарим Андрея Красичкова за сообщение о данном дефекте. (CVE-2018-10915)

- Исправление поведения `INSERT ... ON CONFLICT UPDATE` с более сложным представлением, чем просто `SELECT * FROM ...` (Дин Рашид, Амит Ланготе)

  Ошибочное развёртывание изменяемого представления могло приводить к сбоям или ошибкам «attribute ... has the wrong type» (атрибут ... имеет неверный тип), если список `SELECT` представления не соответствовал в точности списку столбцов нижележащей таблицы. Более того, развив атаку через эту уязвимость, злоумышленник мог изменять столбцы, не имея для них права `UPDATE`, но имея права `INSERT` и `UPDATE` для некоторых других столбцов в таблице. Любой пользователь также мог эксплуатировать её для раскрытия содержимого памяти сервера. (CVE-2018-10925)

- Обеспечение своевременного изменения полей `relfrozenxid` и `relminmxid` в «зафиксированных» системных каталогах (Андрес Фройнд)

  Из-за чрезмерно оптимистичного кеширования эти изменения могли оказаться невидимыми для других сеансов, что приводило к случайным ошибкам и/или разрушению данных. Для общих каталогов, таких как `pg_authid`, проблема была ещё острее, так как устаревшие кешированные данные могли попадать в новые сеансы и оставаться в существующих.

- Исправление поведения в ситуации, когда недавно повышенный сервер аварийно завершается до окончания обработки первой контрольной точки после восстановления (Микаэль Пакье, Кётаро Хоригути, Паван Деоласи, Альваро Эррера)

  В такой ситуации сервер не считал, что при последующем воспроизведении WAL было достигнуто согласованное состояние базы данных, и это препятствовало его перезапуску.

- Исключение выдачи фантомной записи WAL при переработке полностью нулевой страницы индекса btree (Амит Капила)

  Эта ошибка проявлялась в сбоях проверочных утверждений и могла приводить к неоправданным отменам запросов на серверах горячего резерва.

- При воспроизведении WAL добавлена защита от некорректной длины записи, превышающей 1 ГБ (Микаэль Пакье)

  Подобные данные должны считаться некорректными. Ранее код пытался выделить такой объём и получал критическую ошибку, что делало восстановление невозможным.

- При завершении восстановления запись в файл истории линии времени должна откладываться, насколько это возможно (Хейкки Линнакангас)

  Это позволяет избежать ситуаций с несогласованностью состояния линии времени на диске, возникавших при сбое в процессе очистки после восстановления (например, при проблеме с файлом двухфазного состояния).

- Увеличение скорости воспроизведения WAL для транзакций, удаляющих множество отношений (Фудзии Масао)

  В результате этого изменения уменьшается число сканирований общих буферов, поэтому эффект от данного изменения тем больше, чем больше значение shared_buffers.

- Увеличение скорости освобождения блокировки при воспроизведении WAL ведомым сервером (Томас Манро)

- Обеспечение согласованной передачи состояния логической трансляции передатчиками WAL (Саймон Риггс, Савада Масахико)

  Ранее код не мог правильно определить, удалось ли нагнать вышестоящий сервер.

- Обеспечение наличия снимка при выполнении функций ввода типа данных в подписчиках логической репликации (Мин-Цюань Тран, Альваро Эррера)

  Отсутствие снимка в некоторых случаях приводило к сбоям, например, при вводе доменов с ограничениями, определяемыми функциями на языке SQL.

- Устранение дефектов в процедуре обработки снимков при логическом декодировании, приводивших к ошибкам декодирования в редких случаях (Арсений Шер, Альваро Эррера)

- Добавление обработки подтранзакций в рабочих процессах синхронизации таблиц при логической репликации (Амит Хандекар, Роберт Хаас)

  Ранее синхронизация таблицы могла выполняться некорректно в случае отката подтранзакций после модификации синхронизируемой таблицы.

- Обеспечение корректного сброса кешированного списка индексов таблицы после сбоя при создания индекса (Питер Гейган)

  Ранее в этом списке могли оставаться OID нерабочих индексов, что могло приводить к проблемам в том же сеансе.

- Исправление некорректной обработки пустых несжатых страниц списка идентификаторов в индексах GIN (Шивасубраманьян Рамасубраманьян, Александр Коротков)

  Это могло привести к сбою проверочного утверждения после выполнения pg_upgrade для индексов GIN версии до 9.4 (в 9.4 и последующих версиях такие страницы не создаются).

- Выравнивание массивов безымянных семафоров POSIX для уменьшения разделения линий кеша (Томас Манро)

  Это снижает конкуренцию в многопроцессорных системах и устраняет падение производительности (наблюдаемое в сравнении с предыдущими выпусками) в Linux и FreeBSD.

- Обеспечение реакции на сигналы в процессе, производящем параллельное сканирование индекса (Амит Капила)

  Ранее параллельные исполнители могли «зависать», ожидая блокировки страницы индекса, и не замечать сигналы прерывания запроса.

- Обеспечение реакции на сигналы в процессе `VACUUM` в цикле удаления страниц B-дерева (Андрес Фройнд)

  С испорченными индексами btree цикл мог оказаться бесконечным, и прервать его можно было только аварийно отключив сервер.

- Исправление ошибки с оценкой стоимости соединения по хешу при оптимизации inner_unique (Дэвид Роули)

  Это могло приводить к выбору неудачного плана запроса, когда применялась эта оптимизация.

- Исправление некорректной оптимизации классов эквивалентности со столбцами составного типа (Том Лейн)

  Вследствие данной ошибки сервер не понимал, что индекс по составному столбцу может обеспечить порядок сортировки, необходимый для соединения слиянием с данным столбцом.

- Исправление планировщика с целью устранения ошибок «ORDER/GROUP BY expression not found in targetlist» (выражение ORDER/GROUP BY не найдено в целевом списке) при выполнении некоторых запросов с функциями, возвращающими множества (Том Лейн)

- Исправление обработки ключей секционирования с типами данных, которые используют полиморфный класс операторов btree; например, это касается массивов (Амит Ланготе, Альваро Эррера)

- Исправление синтаксиса конструкции SQL-стандарта `FETCH FIRST`, чтобы она принимала (`$n`), как того требует стандарт (Эндрю Гирт)

- Удаление недокументированного ограничения, не допускающего дублирования столбцов в ключах разбиения (Юго Нагата)

- Недопущение использования временных таблиц в качестве секций не временных, а обычных таблиц (Амит Ланготе, Микаэль Пакье)

  Ранее такое использование разрешалось, но работало ненадёжно.

- Исправление в `EXPLAIN` учёта использования ресурсов, в частности, обращений к буферам (Амит Капила, Томас Манро)

- Исправление поведения `SHOW ALL`, чтобы эта команда показывала все параметры членам роли `pg_read_all_settings` и позволяла этим пользователям видеть имя файла и номер строки в представлении `pg_settings` (Лоренц Альбе, Альваро Эррера)

- Корректное добавление схемы к именам ряда объектов в выводе `getObjectDescription` и `getObjectIdentity` (Кётаро Хоригути, Том Лейн)

  Имена правил сортировки, преобразований, объектов текстового поиска, отношений публикаций и объектов расширенной статистики не дополнялись схемой, тогда как должны были.

- Исправление проверки типа `CREATE AGGREGATE` с тем, чтобы к агрегатам с переменными аргументами можно было присоединять функции поддержки распараллеливания (Алексей Баштанов)

- Расширение счётчика строк в команде `COPY FROM` с 32 до 64 бит (Дэвид Роули)

  Тем самым были решены две проблемы с входными данными, содержащими более 4G строк: `COPY FROM WITH HEADER` терял одну строку на каждые 4G строк (не только первую), а в сообщениях об ошибках выводилось неверное количество строк.

- Восстановление возможности удаления слотов репликации в однопользовательском режиме (Альваро Эррера)

  Она была непреднамеренно утрачена в выпуске 10.0.

- Исправление некорректных результатов функции `variance(int4)` и связанных агрегатов при работе в параллельном режиме агрегирования (Дэвид Роули)

- Внесение корректив в обработку узлов `TEXT` и `CDATA` в выражениях столбцов `xmltable()` (Маркус Винанд)

- Защита от возможной ошибки в функции OpenSSL `RAND_bytes()` (Дин Рашид, Микаэль Пакье)

  При редких обстоятельствах это упущение могло приводить к отказам с сообщением «could not generate random cancel key» (не удалось сгенерировать случайный ключ отмены), устранить которые можно было, только перезапустив главный процесс.

- Исправление поведения libpq в некоторых случаях, когда задаётся `hostaddr` (Хари Бабу, Том Лейн, Роберт Хаас)

  Функция `PQhost()` в некоторых случаях выдавала некорректные или вводящие в заблуждение результаты. Теперь она единообразно возвращает имя узла (если оно указано), или адрес узла (если указан только он), или имя узла по умолчанию (обычно `/tmp` либо `localhost`, если опущены оба параметра).

  Также при проверке сертификата SSL с именем сервера могло сравниваться неправильное значение.

  Кроме того, поле с именем узла в `~/.pgpass` могло сравниваться с неправильным значением. Теперь это поле сравнивается с именем сервера (если оно указано), либо с его адресом (если указан только он), либо с `localhost` (если опущены оба параметра).

  Помимо этого, когда значение `hostaddr` нельзя было разобрать, выдавалось неправильное сообщение об ошибке.

  А когда параметры `host`, `hostaddr` или `port` содержат списки через запятую, libpq теперь аккуратнее обрабатывает пустые элементы списков (воспринимая их как значения по умолчанию).

- Добавление в библиотеку ecpg `pgtypes` функции освобождения строк для исключения проблем с управлением памятью в разных модулях (Такаюки Цунакава)

  В Windows могут происходить сбои, если вызов `free` для некоторого блока памяти производится не из той DLL, которая выделила память (с помощью `malloc`). Библиотека `pgtypes` иногда возвращает строки, которые должен освободить вызывающий код, и сделать это корректно оказывается невозможно. Добавленная функция `PGTYPESchar_free()` является просто обёрткой `free`, позволяющей приложениям произвести освобождение по правилам.

- Исправление поддержки в ecpg переменных `long long` в Windows, а также на других платформах, где `strtoll`/`strtoull` объявлены нестандартно или не объявлены вовсе (Хыонг Дангминь, Том Лейн)

- Исправление ошибки с определением типа SQL-оператора в PL/pgSQL, когда изменение правила приводило к изменению семантики оператора в рамках сеанса (Том Лейн)

  Эта ошибка приводила к сбоям проверочных утверждений или, в редких случаях, к тому, что указание `INTO STRICT` не работало должным образом.

- Исправление запроса пароля в клиентских программах, чтобы отображение корректно отключалось в Windows, когда `stdin` — не терминал (Мэтью Стикни)

- Доработка исправления некорректного заключения в кавычки значений для переменных GUC со списками при формировании дампа (Том Лейн)

  Предыдущее исправление для заключения в кавычки значения `search_path` и других переменных со списками в выводе pg_dump, как оказалось, привело к ошибкам со списками с пустыми элементами и риску потери части длинных файловых путей.

- В pg_dump устранено упущение, когда свойства `REPLICA IDENTITY` для индексов ограничений не выгружались (Том Лейн)

  Уникальные индексы, созданные вручную, помечались корректно, а индексы, созданные ограничениями `UNIQUE` или `PRIMARY KEY` — нет.

- В pg_upgrade теперь правильно проверяется, что старый сервер был выключен штатно (Брюс Момджян)

  Ранее проверка могла сработать некорректно при выключении сервера в незамедлительном режиме.

- Исправление в `contrib/hstore_plperl` просмотра скалярных ссылок Perl и предотвращение сбоя в случае, если не удаётся найти хеш-ссылку там, где она ожидается (Том Лейн)
- Устранение сбоя в функции модуля `contrib/ltree` при получении на вход пустого массива (Пьер Дюкроке)
- Исправление обработки ошибок в ряде мест, где мог выдаваться некорректный код ошибки (Микаэль Пакье, Том Лейн, Магнус Хагандер)
- Переоформление скриптов Makefile для обеспечения компоновки программ с собранными в том же дереве библиотеками (например, `libpq.so`), а не с теми, что могут уже находиться в системных каталогах библиотек (Том Лейн)

  Это позволяет избежать проблем при сборке на платформах, где представлены старые версии библиотек PostgreSQL.
- Обновление данных часовых поясов до версии tzdata 2018e, включающее изменение правил перехода на летнее время в Северной Корее, а также корректировку исторических данных для Чехословакии.

  В это обновление вошло переопределение «летнего времени», которое действует в Ирландии, а также действовало в прошлом в Намибии и в Чехословакии. В этих юрисдикциях стандартное время действует летом, а время для экономии света — зимой, поэтому при переходе на это время сдвиг производится на час назад, а не на час вперёд. Это не влияет ни на фактическое смещение от UTC, ни на используемое сокращение часового пояса; эта особенность проявляется лишь в том, что в таких случаях столбец `is_dst` в представлении `pg_timezone_names` будет содержать true для зимнего периода, и false для летнего.

E.46. Выпуск 10.4

Дата выпуска: 2018-05-10

В этот выпуск вошли различные исправления, внесённые после версии 10.3. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.46.1. Миграция на версию 10.4

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Однако если вы используете расширение `adminpack`, его нужно обновить, следуя указаниям в первой записи в списке изменений.

Кроме того, если вас касаются ошибки с пометкой функций, описанные во второй и третьей записи ниже, потребуются дополнительные действия для исправления каталогов баз данных.

Если вы обновляете сервер с более ранней версии, чем 10.3, см. также Раздел E.47.

E.46.2. Изменения

- Лишение роли public права на выполнение функции `pg_logfile_rotate()` из `contrib/adminpack` (Стивен Фрост)

  Функция `pg_logfile_rotate()` является устаревшей оболочкой функции ядра `pg_rotate_logfile()`. Когда в последней была удалена жёсткая проверка суперпользователя, чтобы доступ к этой функции ограничивался правами SQL, эти изменения следовало отразить и в `pg_logfile_rotate()`, но это было упущено. Таким образом, с установленным расширением `adminpack` любой пользователь мог запросить прокрутку файла журнала, что можно считать некритичным нарушением безопасности.

  После установки этого обновления администраторы должны обновить `adminpack`, выполнив `ALTER EXTENSION adminpack UPDATE` в каждой базе данных, где установлен `adminpack`. (CVE-2018-1115)

- Исправление некорректных пометок изменчивости для нескольких встроенных функций (Томас Манро, Том Лейн)

  Функции `query_to_xml`, `cursor_to_xml`, `cursor_to_xmlschema`, `query_to_xmlschema` и `query_to_xml_and_xmlschema` должны были считаться изменчивыми, так как они выполняют пользовательские запросы, в которых могут быть изменчивые операции. Однако они не были помечены должным образом, что было чревато неправильной оптимизацией запросов. Это было исправлено для новых инсталляций в результате корректировки исходных данных каталога, но в существующих инсталляциях ошибочные пометки сохранятся. При практическом использовании этих функций риск кажется небольшим, но в случае необходимости их можно исправить, вручную изменив записи этих функций в `pg_proc`. Например: `ALTER FUNCTION pg_catalog.query_to_xml(text, boolean, boolean, text) VOLATILE`. (Заметьте, что это нужно будет проделать в каждой базе данных инсталляции.) Также вы можете обновить базу до версии с корректными исходными данными, воспользовавшись pg_upgrade.

- Исправление некорректных пометок параллельно-безопасности для нескольких встроенных функций (Thomas Munro, Tom Lane)

  Функции `brin_summarize_new_values`, `brin_summarize_range`, `brin_desummarize_range`, `gin_clean_pending_list`, `cursor_to_xml`, `cursor_to_xmlschema`, `ts_rewrite`, `ts_stat`, `binary_upgrade_create_empty_extension` и `pg_import_system_collations` должны были считаться параллельно-небезопасными, так как одни из них непосредственно модифицируют базу данных, а другие выполняют пользовательские запросы, которые могут это делать. Они были некорректно помечены как параллельно-безопасные, что могло приводить к неожиданным ошибкам запросов. Это было исправлено для новых инсталляций в результате корректировки исходных данных каталога, но в существующих инсталляциях ошибочные пометки сохранятся. При практическом использовании этих функций риск кажется небольшим, если только не включён режим `force_parallel_mode`, но в случае необходимости их можно исправить, вручную изменив записи этих функций в `pg_proc`. Например: `ALTER FUNCTION pg_catalog.brin_summarize_new_values(regclass) PARALLEL UNSAFE`. (Заметьте, что это нужно будет проделать в каждой базе данных инсталляции.) Также вы можете обновить базу до версии с корректными исходными данными, воспользовавшись pg_upgrade.

- Предотвращение повторного использования для TOAST идентификаторов OID, соответствующих уже неактуальным, но ещё не очищенным записям TOAST (Паван Деоласи)

  После зацикливания счётчика OID имеется возможность использования для значения TOAST идентификатора OID, соответствующего ранее удалённой записи в той же таблице TOAST. Если запись не была очищена к тому времени, это приводило к ошибкам «unexpected chunk number 0 (expected 1) for toast value *nnnnn*» (неожиданный номер порции 0 (ожидался 1) для значения TOAST *nnnnn*), которые сохранялись до удаления неактуальной записи командой `VACUUM`. В качестве решения выбор таких OID при создании новых записей TOAST теперь исключается.

- Корректная проверка всех ограничений `CHECK` для отдельных секций при выполнении `COPY` в секционированную таблицу (Эцуро Фудзита)

  Ранее проверялись только ограничения, заданные для секционированной таблицы в целом.

- Допущение значений `TRUE` и `FALSE` в качестве границ секции (Амит Ланготе)

  Ранее для логического секционирующего столбца принимались только строковые литералы, но pg_dump выводит эти значения как `TRUE`/`FALSE`, что приводило к ошибкам при выгрузке/загрузке данных.

- Исправление работы с памятью в функциях сравнения ключей секционирования (Альваро Эррера, Амит Ланготе)

  Эта ошибка могла приводить к сбоям при использовании для ключей секционирования классов операторов, определённых пользователем.

- Устранение возможности сбоя при добавлении запросом кортежей в несколько секций секционированной таблицы, когда эти секции имеют разные типы строк (Эцуро Фудзита, Амит Ланготе)
- Изменение алгоритма ANALYZE в части модификации pg_class.reltuples (Дэвид Гулд)

  Ранее плотность кортежей могла обновляться только для страниц, прошедших обработку ANALYZE, для остальных плотность считалась прежней. В большой таблице, где ANALYZE выбирает лишь небольшой процент страниц, это означает, что возможно лишь незначительное изменение общей оценки плотности кортежей, и поэтому reltuples будет меняться практически пропорционально изменениям физического размера таблицы (relpages) вне зависимости от того, что фактически происходит в таблице. В результате может наблюдаться настолько большое увеличение reltuples по сравнению с реальностью, что автоматическая очистка практически отключается. Для исправления этой ошибки принимается, что выборка ANALYZE является статистически несмещённой (какой она и должна быть), и наблюдаемая в ней плотность просто экстраполируется на всю таблицу.
- Включение объектов расширенной статистики в набор свойств таблицы, копируемых конструкцией CREATE TABLE ... LIKE ... INCLUDING ALL (Дэвид Роули)

  Также добавлено указание INCLUDING STATISTICS для явного управления включением этой информации.
- Исправление работы CREATE TABLE ... LIKE со столбцами идентификации типа bigint (Питер Эйзентраут)

  На платформах, где long имеет размер 32 бита (что включает 64-битные Windows, а также большинство 32-битных систем), копируемые параметры последовательностей усекались до 32 бит.
- Предупреждение взаимоблокировок в параллельных командах CREATE INDEX CONCURRENTLY, выполняемых на уровнях изоляции SERIALIZABLE и REPEATABLE READ (Том Лейн)
- Устранение возможности замедленного выполнения REFRESH MATERIALIZED VIEW CONCURRENTLY (Томас Манро)
- Устранение ошибки в UPDATE/DELETE ... WHERE CURRENT OF в случае, когда задействованный курсор использует план сканирования только индекса (Юго Нагата, Том Лейн)
- Исправление некорректного планирования, когда предложения соединения передавались в параметризованные пути (Эндрю Гирт, Том Лейн)

  Эта ошибка могла привести к неправильной классификации условия как «фильтра соединения» для внешнего соединения, тогда как оно должно быть простым «фильтром», и в итоге мог получиться некорректный результат соединения.
- Устранение возможности некорректного построения плана сканирования только индекса в случаях, когда один столбец таблицы фигурирует в нескольких индексах, но не во всех этих индексах используются классы операторов, которые могут выдать значение столбца (Кётаро Хоригути)
- Исправление некорректной оптимизации ограничений CHECK с гарантированными NULL-подвыражениями в условиях верхнего уровня AND/OR (Том Лейн, Дин Рашид)

  В результате, например, ограничение-исключение могло исключить из запроса дочернюю таблицу, которая не должна быть исключена.
- Предотвращение сбоя планировщика в случаях, когда запрос содержит несколько наборов GROUPING SETS, и ни один из них нельзя получить сортировкой (Эндрю Гирт)
- Устранение сбоя исполнителя в результате двойного освобождения памяти с некоторыми вариантами использования GROUPING SET (Питер Гейган)
- Исправление некорректного выполнения замкнутого соединения переходных таблиц (Томас Манро)

- Предотвращение сбоя в случае, когда триггер события перезаписи таблицы добавляется одновременно с выполнением команды, которая может вызвать такой триггер (Альваро Эррера, Эндрю Гирт, Том Лейн)
- Предотвращение ошибки при прерывании запроса или прекращении сеанса в момент фиксирования подготовленной транзакции (Стас Кельвич)
- Ликвидация утечки памяти на время выполнения запроса в последовательно выполняемых соединениях по хешу (Том Лейн)
- Устранение возможных утечек или двойного освобождения закреплений буферов карты видимости (Амит Капила)
- Устранение неоправданной пометки страниц как полностью видимых (Дэн Вуд, Паван Деоласи, Альваро Эррера)

  Это могло происходить при блокировании (но не удалении) некоторых кортежей. Хотя запросы будут продолжать выполняться корректно, процедура очистки обычно игнорирует такие страницы, вследствие чего кортежи на них никогда не будут замораживаться. В последних выпусках это в конце концов приведёт к появлению ошибок вида «found multixact *nnnnn* from before relminmxid *nnnnn*» (найдена мультитранзакция *nnnnn*, предшествующая relminmxid *nnnnn*).
- Исправление излишне строгой проверки в `heap_prepare_freeze_tuple` (Альваро Эррера)

  Это могло приводить к необоснованной ошибке «cannot freeze committed xmax» (не удаётся заморозить зафиксированный xmax) в базах данных, обновлённых с помощью pg_upgrade с версии 9.2 или старее.
- Устранение потери указателя в случаях, когда написанный на C триггер, выполняемый до изменения строки, возвращает старый кортеж («old») (Рушаб Латиа)
- Понижение уровня блокировки при планировании работы автоочистки (Джефф Джейнс)

  Предыдущее поведение создавало значительные препятствия для параллельного выполнения рабочих процессов в базах, содержащих множество таблиц.
- Обеспечение копирования имени клиентского компьютера при копировании данных `pg_stat_activity` в локальную память (Эдмунд Хорнер)

  Ранее предположительно локальный экземпляр содержал указатель на разделяемую память, вследствие чего содержимое поля с именем клиентского компьютера могло неожиданно измениться при отключении какого-либо сеанса.
- Корректная обработка информации `pg_stat_activity` для вспомогательных процессов (Эдмунд Хорнер)

  Поля `application_name`, `client_hostname` и `query` для таких процессов могли содержать неверные данные.
- Исправление некорректной обработки нескольких составных аффиксов в словарях `ispell` (Артур Закиров)
- Исправление поиска (то есть сканирования индекса с операторами неравенства), зависимого от правила сортировки, в индексах SP-GiST, построенных по текстовым столбцам (Том Лейн)

  Такой поиск мог возвращать неправильный набор строк для большинства правил сортировки, отличных от C.
- Предотвращение утечки памяти на время выполнения запроса в классах операторов SP-GiST, использующих переходящие значения (Антон Дигнос)
- Корректировка вычисления количества кортежей в индексе при изначальном построении индекса SP-GiST (Томаш Вондра)

  Ранее количество кортежей в индексе считалось равным количеству кортежей в нижележащей таблице, что неверно в случае частичного индекса.

- Корректировка вычисления количества кортежей в индексе при очистке индекса GiST (Андрей Бородин)

  Ранее оно считалось равным примерному количеству кортежей в куче, что провоцировало неточность и определённо было ошибочным в случае частичного индекса.

- Исправление поведения в особом случае, когда ведомый реплицирующий сервер «застревал» на записи продолжения WAL (Кётаро Хоригути)

- При логическом декодировании приняты меры во избежание двойной обработки данных WAL при перезапуске передатчика WAL (Крейг Рингер)

- Исправление механизма логической репликации, чтобы он не полагался на совпадение OID типов между локальным и удалённым серверами (Масахико Савада)

- Поддержка использования `scalarltsel` и `scalargtsel` с расширенными типами данных (Томаш Вондра)

- Уменьшение потребления памяти libpq в случаях, когда сервер выдаёт ошибку после получения большого объёма результата запроса (Том Лейн)

  Полученный ранее результат должен быть отброшен до, а не после обработки ошибки. На некоторых платформах, в частности в Linux это может влиять на то, сколько памяти будет занимать приложение.

- Устранение сбоев в ecpg, вызванных двойным освобождением памяти (Патрик Крекер, Дживан Ладхе)

- Исправление обработки переменных `long long int` в ecpg, собранном с использованием MSVC (Михаэль Мескес, Эндрю Гирт)

- Исправление некорректного заключения в кавычки значений для переменных GUC со списками при формировании дампа (Микаэль Пакье, Том Лейн)

  Переменные `local_preload_libraries`, `session_preload_libraries`, `shared_preload_libraries` и `temp_tablespaces` не заключались в кавычки корректно в выводе pg_dump. Это могло приводить к проблемам, если эти переменным присваивались значения в конструкциях `CREATE FUNCTION ... SET` или `ALTER DATABASE/ROLE ... SET`.

- Предотвращение отказа pg_recvlogical при подключении к серверам PostgreSQL до 10 версии (Микаэль Пакье)

  В результате предыдущего исправления программа pg_recvlogical, не проверяя версию сервера, выдавала команду, которая должна предназначаться только серверам версии 10 и новее.

- Исправление поведения pg_rewind, чтобы на целевом сервере удалялись файлы, которые могли быть удалены в процессе выполнения на исходном сервере (Такаюки Цунакава)

  В противном случае на целевом сервере могла нарушаться согласованность данных, особенно если это был файл сегмента WAL.

- Исправление в pg_rewind обработки таблиц в дополнительных табличных пространствах (Такаюки Цунакава)

- Исправление обработки целочисленного переполнения в циклах `FOR` на языке PL/pgSQL (Том Лейн)

  Ранее с некоторыми компиляторами, отличными от gcc, переполнение переменной цикла не проверялось, в результате чего цикл оказывался бесконечным.

- Исправление регрессионных тестов PL/Python для совместимости с Python 3.7 (Питер Эйзентраут)

- Поддержка тестирования PL/Python и связанных модулей при сборке с использованием Python 3 и MSVC (Эндрю Дунстан)

- Устранение ошибок при изначальном построении индексов `contrib/bloom` (Томаш Вондра, Том Лейн)

  Предотвращение возможного исчезновения последнего кортежа таблицы из индекса. Исправление подсчёта количества кортежей в частичных индексах.

- Переименование функций `b64_encode` и `b64_decode` во избежание конфликта со встроенными функциями Solaris 11.4 (Райнер Орт)

- Синхронизация нашей копии библиотеки timezone с выпущенной IANA версией 2018e (Том Лейн)

  В этой версии усовершенствован компилятор данных часовых поясов zic для работы с отрицательными смещениями при переходе на летнее время. Хотя проект PostgreSQL в настоящее время не поставляет такие данные часовых поясов, zic может применяться с данными, полученными непосредственно от IANA, поэтому кажется разумным обновить zic сейчас.

- Обновление данных часовых поясов до версии tzdata 2018d, включающее изменения правил перехода на летнее время в Палестине и Антарктиде (станция Кейси), плюс корректировку исторических данных для Португалии и её колоний, а также Уругвая и островов Эндербери, Ямайка, Теркс и Кайкос.

E.47. Выпуск 10.3

Дата выпуска: 2018-03-01

В этот выпуск вошли различные исправления, внесённые после версии 10.2. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.47.1. Миграция на версию 10.3

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Однако если в вашей СУБД не все пользователи взаимно доверяют друг другу либо если вы поддерживаете приложение или расширение, предназначенное для использования в произвольных ситуациях, настоятельно рекомендуется прочитать об изменениях в первой записи ниже и предпринять соответствующие действия для защиты вашей инсталляции или кода.

Также изменения, описанные во втором пункте списка ниже, могут приводить к сбоям функций, используемых в индексных выражениях или материализованных представлениях, во время автоматического анализа или при восстановлении выгруженных данных. Поэтому после обновления проверьте журнал сервера на предмет подобных проблем и исправьте затронутые функции.

Также, если вы обновляете сервер с более ранней версии, чем 10.2, см. Раздел E.48.

E.47.2. Изменения

- Добавление в документацию информации о настройке серверов и приложений для защиты от атак с внедрением троянского кода через путь поиска (Ной Миш)

  Когда используется значение `search_path`, включающее схемы, доступные для записи злонамеренному пользователю, он может перехватывать управление над выполнением запросов и затем запускать произвольный SQL-код с правами атакуемого пользователя. Хотя есть возможность составлять запросы, защищённые от подобного перехвата, это требует кропотливой работы, и при этом очень легко что-то упустить. Поэтому мы рекомендуем использовать конфигурации, в которых пути поиска не могут включать недоверенные схемы. Соответствующая информация добавлена в Подраздел 5.8.6 (для пользователей и администраторов баз данных), Раздел 31.1 (для разработчиков приложений), Подраздел 35.15.6 (для разработчиков расширений) и CREATE FUNCTION (для разработчиков функций с характеристикой `SECURITY DEFINER`). (CVE-2018-1058)

- Предотвращение использования небезопасных значений `search_path` в pg_dump и других клиентских программах (Ной Миш, Том Лейн)

  pg_dump, pg_upgrade, vacuumdb и другие приложения, предоставляемые PostgreSQL, были уязвимы для перехвата выполнения, описанного в предыдущем пункте; так как эти приложения обычно запускаются суперпользователями, они представляли собой привлекательные средства для атаки. Чтобы защитить их вне зависимости от того, защищена ли вся инсталляция в целом, они были изменены так, чтобы для них параметр `search_path` содержал только `pg_catalog`. Теперь это так же касается и рабочих процессов автоочистки.

  В случаях, когда этими программами неявно вызываются определённые пользователем функции — например, это могут быть пользовательские функции в выражениях индексов — более строгое значение `search_path` может приводить к ошибкам, которые потребуется исправить так, чтобы эти функции никак не зависели от пути поиска, с которым они выполняются. Это всегда рекомендовалось делать, но сейчас это необходимо для корректного поведения. (CVE-2018-1058)

- Предотвращение попыток логической репликации передавать изменения в непубликуемых отношениях (Питер Эйзентраут)

  Публикация всех таблиц (с пометкой `FOR ALL TABLES`) могла некорректно передавать изменения в материализованных представлениях и таблицах `information_schema`, которые не должны попадать в поток изменений.

- Исправление некорректного поведения перепроверок одновременных изменений со ссылками CTE, фигурирующими во внутренних планах (Том Лейн)

  Если ссылка на CTE (содержимое предложения `WITH`) использовалась в InitPlan или SubPlan, и запросу требовалось провести перепроверку из-за попытки изменения или блокировки одновременно изменяемой строки, могли быть получены неверные результаты.

- Устранение сбоев планировщика при перекрывающихся предложениях соединения слиянием во внешнем соединении (Том Лейн)

  Эти дефекты приводили в особых случаях к ошибкам планировщика «left and right pathkeys do not match in mergejoin» (нарушено соответствие левых и правых ключей пути в соединении слиянием) или «outer pathkeys do not match mergeclauses» (внешние ключи пути не соответствуют предложениям слияния).

- Исправление ошибки в pg_upgrade, когда не удавалось сохранить `relfrozenxid` для материализованных представлений (Том Лейн, Андрес Фройнд)

  Данное упущение могло приводить к разрушению данных в материализованных представлениях после обновления. Это могло проявляться в ошибках «could not access status of transaction» (не удалось получить состояние транзакции) или «found xmin from before relfrozenxid» (найден xmin перед relfrozenxid). Эта проблема была более вероятна в редко обновляемых материализованных представлениях или в представлениях, обновляемых только командой `REFRESH MATERIALIZED VIEW CONCURRENTLY`.

  Если такое разрушение имело место, его можно исправить, обновив материализованное представление (без указания `CONCURRENTLY`).

- Исправление некорректного вывода pg_dump для некоторых граничных значений последовательностей (Алексей Баштанов)

- Исправление неправильной обработки в pg_dump объектов `STATISTICS` (Том Лейн)

  Схема объекта расширенной статистики неправильно помечалась в оглавлении таблицы выгружаемых данных, что могло приводить к некорректным результатам при восстановлении отдельных схем. Также некорректно восстанавливалась информация о владельце. Кроме того, логика работы изменена так, чтобы объекты статистики выгружались/восстанавливались или нет как независимые объекты, без учёта того, производится ли выгрузка/восстановление

таблицы, к которой они привязаны. Исходное определение не должно распространяться на планируемое будущее расширение статистики нескольких таблиц.

- Исправление некорректной обработки имён функций PL/Python в стеках ошибки `CONTEXT` (Том Лейн)

  Ошибка, произошедшая во вложенном вызове функции PL/Python (то есть функции, вызванной через SPI-запрос из другой функции PL/Python), могла привести к тому, что в трассировке стека вместо ожидаемого результата имя внутренней функции фигурировало дважды. Также ошибка во вложенном блоке PL/Python `DO` могла привести к обращению по нулевому указателю на некоторых платформах.

- Максимальное значение `log_min_duration` для `contrib/auto_explain` доведено до `INT_MAX`, что составляет около 24 суток вместо 35 минут (Том Лейн)

- Для разнообразных переменных GUC добавлена пометка `PGDLLIMPORT` в целях облегчения переноса модулей расширений в Windows (Метин Дослу)

E.48. Выпуск 10.2

Дата выпуска: 2018-02-08

В этот выпуск вошли различные исправления, внесённые после версии 10.1. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.48.1. Миграция на версию 10.2

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Однако если вы используете оператор `~>` расширения `contrib/cube`, прочитайте ниже запись о нём.

Также, если вы обновляете сервер с более ранней версии, чем 10.1, см. Раздел E.49.

E.48.2. Изменения

- Исправление обработки ключей секционирования, содержащих несколько выражений (Альваро Эррера. Дэвид Роули)

  Эта ошибка приводила к сбоям или, со специально сконструированными данными, к раскрытию произвольного содержимого памяти обслуживающего процесса. (CVE-2018-1052)

- Временные файлы, создаваемые программой pg_upgrade, должны быть недоступны для чтения всеми пользователями (Том Лейн, Ной Миш)

  Программа pg_upgrade обычно ограничивает доступ к своим временным файлам, чтобы их мог читать и записывать только запускающий её пользователь. Но временные файлы, содержащие вывод `pg_dumpall -g` могли быть доступны для чтения группе или всем пользователям, а возможно и для записи, если это допускало значение `umask` этого пользователя. При типичном использовании в многопользовательских системах `umask` и/или разрешения в рабочем каталоге достаточно жёсткие и эта проблема неактуальна; но pg_upgrade может использоваться и в сценариях, где это упущение могло привести, например, к утечке паролей баз данных. (CVE-2018-1053)

- Исправление очистки кортежей, которые были изменены при установленной блокировке разделяемого ключа (Андрес Фройнд, Альваро Эррера)

  В некоторых случаях операция `VACUUM` не удаляла такие кортежи, даже когда они теряли актуальность, что приводило к различным сценариям повреждения данных.

- Исправление упущения, когда метастраница хеш-индекса не помечалась как «грязная» после добавления новой страницы переполнения, что могло приводить к порче индекса (Лисянь Цзоу, Амит Капила)

- Осуществление очистки очереди добавлений индекса GIN при выполнении VACUUM (Масахико Савада)

  Это необходимо для гарантированного удаления неактуальных записей индекса. Прежний код делал это в другом порядке, позволяя операции VACUUM пропускать очистку, если какие-то другие процессы производили очистку параллельно. Однако при этом в индексе могли оставаться некорректные записи.

- Исправление неправильной блокировки буфера при некоторых чтениях LSN (Якоб Чемпион, Асим Правин, Ашвин Агравал)

  Эти ошибки могли приводит к некорректному поведению при многопоточной нагрузке. Потенциальные последствия не были характеризованы полностью.

- Исправление некорректных результатов запросов в случаях, когда происходило упрощение подзапросов, результаты которых использовались в `GROUPING SETS` (Хейкки Линнакангас)

- Исправление обработки ограничений секционирования по списку с ключами секционирования логического типа и массивами (Амит Ланготе)

- Исключение неоправданной ошибки в запросе с деревом наследования, который выполнялся одновременно с тем, как некоторая дочерняя таблица удалялась из этого дерева командой `ALTER TABLE NO INHERIT` (Том Лейн)

- Устранение разнообразных ошибок взаимоблокировки при выполнении в нескольких сеансах команды `CREATE INDEX CONCURRENTLY` (Джефф Джейнс)

- Изменение полей размера таблицы в процессе `VACUUM FULL` на более раннем этапе (Амит Капила)

  Это предупреждает неоптимальное поведение при перестраивании хеш-индексов таблицы, так как для расчёта начального размера хеша используется статистика `pg_class`.

- Исправление работы `UNION`/`INTERSECT`/`EXCEPT` с нулём столбцов (Том Лейн)

- Запрет использования столбцов идентификации с типизированными таблицами и секциями (Микаэль Пакье)

  Такое использование теперь будет считаться неподдерживаемым.

- Исправление разнообразных ошибок, когда в столбец идентификации при добавлении записи не вставлялось правильное значение по умолчанию (Микаэль Пакье, Питер Эйзентраут)

  В некоторых контекстах, а именно в `COPY` и `ALTER TABLE ADD COLUMN`, ожидаемое значение по умолчанию не применялось, а вместо него вставлялось значение NULL.

- Устранение ошибок в ситуациях, когда дерево наследования содержит дочерние сторонние таблицы (Эцуро Фудзита)

  Комбинация обычных и сторонних таблиц в дереве наследования провоцировала построение некорректных планов в запросах `UPDATE` и `DELETE`. Это приводило к видимым ошибкам в некоторых случаях, особенно когда в дочерней сторонней таблице присутствовали триггеры уровня строк.

- Исправление дефекта со связанным подзапросом `SELECT` внутри `VALUES` в подзапросе `LATERAL` (Том Лейн)

- Устранение ошибки планировщика «could not devise a query plan for the given query» (не удалось выработать план для данного запроса) в некоторых случаях, включая вложенные `UNION ALL` в подзапросе LATERAL (Том Лейн)

- Реализована возможность использовать статистику функциональной зависимости для логических столбцов (Том Лейн)

  Ранее, хотя расширенную статистику можно было объявить и собирать по логическим столбцам, планировщик не мог её применить.

- Устранение недооценивания количества групп, выдаваемых подзапросами, в которых вызываются функции, возвращающие множества, в группируемых столбцах (Том Лейн)

  В случаях, подобных `SELECT DISTINCT unnest(foo)`, в версии 10.0 приблизительная оценка количества строк стала давать меньшее число по сравнению с предыдущими версиями, что могло приводить к выбору неоптимальных планов. Теперь восстановлен предыдущий алгоритм оценивания.
- Исправление работы триггеров в процессах логической репликации (Петр Желинек)
- Исправление логического декодирования, чтобы файлы на диске корректно очищались от данных сбойных транзакций (Атсуши Торикоши)

  Процедура логического декодирования может выносить записи WAL на диск, если транзакции генерируют много записей WAL. Обычно эти файлы очищаются после фиксирования транзакции или поступления записи о её прерывании; но в отсутствие таких записей код очистки работал неправильно.
- Исправление тайм-аута в процессе walsender и реакции на прерывания при обработке большой транзакции (Петр Желинек)
- Устранение условий гонки при удалении источника репликации, когда ожидание удаляющего процесса могло быть бесконечным (Том Лейн)
- Предоставление членам роли `pg_read_all_stats` возможности видеть статистику walsender в представлении `pg_stat_replication` (Фейке Стинберген)
- Отображение процессов walsender, передающих базовые резервные копии, как активные в представлении `pg_stat_activity` (Магнус Хагандер)
- Исправление обозначения метода аутентификации `scram-sha-256` в представлении `pg_hba_file_rules` (Микаэль Пакье)

  Ранее его название выводилось как `scram-sha256`, что могло смущать пользователей некорректным написанием.
- Добавление в `has_sequence_privilege()` поддержки проверок `WITH GRANT OPTION`, как это сделано в других функциях проверки прав (Джо Конвей)
- В базах данных с кодировкой UTF8 любые XML-объявления, выбирающие другую кодировку, должны игнорироваться (Павел Стехуле, Ной Миш)

  Мы всегда храним документы XML в кодировке базы данных, так что позволяя libxml обрабатывать объявления с другой кодировкой, мы получим ошибочные результаты. Если кодировка базы — не UTF8, мы в любом случае не обещали поддерживать XML-данные с не ASCII-кодировкой, так что предыдущее поведение сохранено для совместимости ошибок. Это изменение затрагивает только `xpath()` и связанные функции; другой код и ранее работал так.
- Обеспечение прямой совместимости с будущими изменениями младших версий протокола (Роберт Хаас, Бадрул Чоудхури)

  Ранее серверы PostgreSQL просто отклоняли запросы на использование версий протокола новее 3.0, так что никакого функционального отличия старших номеров от младших не было. Теперь клиенты могут запрашивать версии 3.x и получать в ответ не отказ, а сообщение, говорящее, что сервер поддерживает только версию 3.0. В данный момент от этого ничего не меняется, но перенос этого изменения в предыдущие версии должен ускорить внедрение небольших усовершенствований протокола в будущем.
- Предоставление возможности клиенту, поддерживающему привязку канала SCRAM, (в частности libpq v11 или новее) подключаться к серверу v10 (Микаэль Пакье)

  Версия 10 не реализует такую функциональность, и согласование её использования производилось некорректно.
- Предотвращение долгоживущих циклов в `ConditionVariableBroadcast()` (Том Лейн, Томас Манро)

При неудачно сложившихся временных интервалах процесс, пытающийся разбудить все спящие процессы по условной переменной, мог зацикливаться на неопределённое время. Вследствие ограниченного использования условных переменных в v10, эта проблема затрагивала только параллельные сканирования индексов и некоторые операции со слотами репликации.

- Корректный сброс ожиданий условных переменных при прерывании подтранзакции (Роберт Хаас)
- Обеспечение своевременного завершения всех дочерних процессов, ожидающих условные переменные, в случае прекращения работы процесса postmaster (Том Лейн)
- Устранение сбоев параллельных процессов при использовании более чем одного узла Gather (Томас Манро)
- Ликвидация зависания в процессе параллельного сканирования индекса при обработке удалённой или наполовину неактуальной страницы индекса (Амит Капила)
- Предотвращение краха в случае, когда при параллельном сканировании битовой карты не удаётся выделить сегмент разделяемой памяти (Роберт Хаас)
- Исправление реакции на сбой при запуске параллельного рабочего процесса (Амит Капила, Роберт Хаас)

  Ранее параллельный запрос мог зависнуть на неопределённое время, если не удавалось запустить рабочий процесс, вследствие ошибки в `fork()` или других редких проблем.
- Исключение неоправданной ошибки, которая выдавалась, когда не удавалось получить параллельные исполнители при запуске параллельного запроса (Роберт Хаас)
- Исправление сбора статистики `EXPLAIN`, получаемой от параллельных исполнителей (Амит Капила, Томас Манро)
- Строки запросов, передаваемые параллельным исполнителям, всегда должны завершаться нулём (Томас Манро)

  Это предотвращает вывод мусора в журнал главного процесса от таких исполнителей.
- Устранение небезопасного предположения о выравнивании при работе с типом `__int128` (Том Лейн)

  Обычно компиляторы полагают, что переменные `__int128` выравниваются по 16-байтовым границам, но наша инфраструктура выделения памяти не готова гарантировать это, а увеличение MAXALIGN кажется неподходящим по множеству причин. В качестве решения код был изменён так, чтобы тип `__int128` можно было использовать только когда мы можем сказать компилятору, что предполагается не такое крупное выравнивание. Единственный замеченный симптом этой проблемы — сбои в некоторых параллельных запросах с агрегированием.
- Предотвращение сбоев с переполнением стека при планировании операций со множествами с крайне большой вложенностью (`UNION`/`INTERSECT`/`EXCEPT`) (Том Лейн)
- Предотвращение краха при перепроверке EvalPlanQual в узле сканирования индекса, являющимся внутренним потомком соединения слиянием (Том Лейн)

  Это могло происходить только при изменении или выборке `SELECT FOR UPDATE` данных соединения, когда одновременно изменялись некоторые выбранные строки.
- Устранение ошибки в процессе автоочистки, когда расширенная статистика для таблицы определена, но не может быть вычислена (Альваро Эррера)
- Ликвидация обращений по нулевому указателю для некоторых типов адресов LDAP, задаваемых в файле `pg_hba.conf` (Томас Манро)
- Предотвращение исчерпания памяти из-за разрастания простых хеш-таблиц (Томаш Вондра, Андрес Фройнд)

- Исправление демонстрационных функций `INSTR()` в документации по PL/pgSQL (Юго Нагата, Том Лейн)

  В документации утверждалось, что эти функции совместимы с Oracle®, но это было не совсем так. В частности, по-разному интерпретировалось отрицательное значение третьего параметра: Oracle воспринимает это значение как последнюю позицию, с которой может начинаться целевая подстрока, а наши функции считали, что это последняя позиция, где строка может заканчиваться. Также Oracle выдаёт ошибку в случае нулевого или отрицательного четвёртого параметра, тогда как наши функции возвращали ноль.

  Код этого примера был изменён для большего соответствия поведению Oracle. Пользователям, которые скопировали этот код в свои приложения, возможно, имеет смысл обновить свои копии.

- Исправление поведения pg_dump, чтобы ACL (разрешения), комментарии и метки безопасности можно было надёжно идентифицировать в архивных выходных форматах (Том Лейн)

  Компонент «метка» записи ACL в архиве обычно представлял собой просто имя целевого объекта. В его начало теперь добавляется тип объекта, чтобы записи ACL соответствовали соглашениям, уже принятым для записей комментариев и меток безопасности. Кроме того, обозначения комментариев и меток безопасности, заданных для собственно базы данных, теперь должны начинаться с `DATABASE`, чтобы они соответствовали тому же соглашению. Это предупреждает ложные срабатывания в коде, который пытается найти записи больших объектов по строкам, начинающимся со слов `LARGE OBJECT`. Прежнее поведение могло приводить к неправильной классификации записей как данные и нежелательным результатам при восстановлении выгруженной только схемы или только данных.

  Заметьте, что следствием этого стало изменение видимого пользователями вывода `pg_restore --list`.

- Переименование функции `copy_file_range` в pg_rewind во избежание конфликта с новым системным вызовом Linux с таким же именем (Андрес Фройнд)

  Это изменение предотвращает ошибки при сборке с новыми версиями glibc.

- В ecpg добавлено выявление массивов индикаторов с неправильной длиной и сообщение об ошибке (Дэвид Рейдер)

- Изменение поведения оператора `cube ~> int` в расширении `contrib/cube` для обеспечения его совместимости с поиском kNN (Александр Коротков)

  Смысл второго аргумента (выбирающего размерность) был изменён, чтобы можно было определённо сказать, какое именно значение выбирается в кубах переменных размерностей.

  Это изменение нарушает совместимость, но так как этот оператор предназначался для поисков kNN, в прежнем виде он был бесполезен. После установки этого обновления все материализованные представления или индексы с выражениями, использующими этот оператор, необходимо обновить/перестроить.

- Предотвращение срабатывания проверки истинности внутри libc в расширении `contrib/hstore` из-за использования `memcpy()` с равными указателями источника и получателя (Томаш Вондра)

- Исправление некорректного отображения битовых карт NULL для кортежей в `contrib/pageinspect` (Максим Милютин)

- Исправление некорректного вывода функции `hash_page_items()` расширения `contrib/pageinspect` (Масахико Савада)

- В расширении `contrib/postgres_fdw` ликвидирована ошибка планировщика «outer pathkeys do not match mergeclauses» (внешние ключи пути не соответствуют предложениям слияния) при построении плана, включающего удалённое соединение (Роберт Хаас)

- В contrib/postgres_fdw предотвращён сбой планировщика при дублировании элементов GROUP BY (Дживан Чок)
- Добавление современных примеров настройки автозапуска Postgres в macOS (Том Лейн)

  Скрипты в contrib/start-scripts/osx используют инфраструктуру, которая устарела десять лет назад, и абсолютно неработоспособны во всех версиях macOS, выпущенных в последние два года. Добавлен новый подкаталог contrib/start-scripts/macos со скриптами, использующими новую инфраструктуру launchd.
- Исправление некорректного выбора зависящих от конфигурации библиотек OpenSSL в Windows (Эндрю Дунстан)
- Поддержка компоновки с версиями libperl, собранными компилятором MinGW (Ной Миш)

  Это позволяет собирать PL/Perl с некоторыми распространёнными дистрибутивами Perl для Windows.
- Исправление в сборке MSVC проверки, требуется ли 32-битной libperl определение -D_USE_32BIT_TIME_T (Ной Миш)

  Имеющиеся дистрибутивы Perl ожидают разного, и при этом нет никакой возможности надёжно проверить это, поэтому пришлось добавить проверку фактического поведения используемой библиотеки во время компиляции.
- В Windows обработчик аварийного завершения должен устанавливаться на более раннем этапе запуска главного процесса (Такаюки Цунакава)

  Это поможет получить дамп памяти при ошибках в те моменты в начале запуска, в которые раньше дамп не записывался.
- В Windows устранены сбои, связанные с преобразованием кодировок при выводе сообщений на самых ранних стадиях запуска процесса postmaster (Такаюки Цунакава)
- Использование нашего ранее написанного кода циклических блокировок для Motorola 68K во OpenBSD, а также в NetBSD (Давид Карлье)
- Добавление поддержки циклических блокировок для Motorola 88K (Давид Карлье)
- Обновление данных часовых поясов до версии tzdata 2018c, включающее изменения правил перехода на летнее время в Бразилии, в Сан-Томе и Принсипи, а также корректировки исторических данных для Боливии, Японии и Южного Судана. Был удалён часовой пояс US/Pacific-New (это был просто псевдоним для пояса America/Los_Angeles).

E.49. Выпуск 10.1

Дата выпуска: 2017-11-09

В этот выпуск вошли различные исправления, внесённые после версии 10.0. За информацией о нововведениях версии 10 обратитесь к Разделу E.50.

E.49.1. Миграция на версию 10.1

Если используется версия 10.X, выгрузка/восстановление базы не требуется.

Однако если вы используете индексы BRIN, прочитайте четвёртую запись в списке изменений.

E.49.2. Изменения

- Обеспечение проверки разрешений на уровне таблиц и политик RLS при выполнении INSERT ... ON CONFLICT DO UPDATE во всех случаях (Дин Рашид)

  Путь изменения данных в команде INSERT ... ON CONFLICT DO UPDATE требовал наличия разрешения SELECT для всех столбцов в решающем индексе, но в случае указания решающего ограничения по имени должная проверка отсутствовала. Кроме того, для таблиц с

включённой защитой на уровне строк не проверялось, соответствуют ли изменённые строки политикам SELECT (вне зависимости от способа задания решающего индекса). (CVE-2017-15099)

- Устранение сбоя при несовпадении типа записи в json{b}_populate_recordset() (Микаэль Пакье, Том Лейн)

  Эти функции использовали тип результата, заданный в предложении FROM ... AS, не проверяя, соответствует ли он фактическому типу поступившего кортежа. В случае несоответствия обычно происходил сбой, хотя также вероятным было раскрытие содержимого памяти сервера. (CVE-2017-15098)

- Исправление скриптов запуска сервера — переключение на $PGUSER до открытия файла $PGLOG (Ной Миш)

  Ранее файл журнала postmaster открывался ещё под именем root. Таким образом, владелец базы данных мог произвести атаку на другого пользователя системы, сделав $PGLOG символической ссылкой на некоторый другой файл, который в результате можно было испортить добавленными в него сообщениями журнала.

  По умолчанию эти скрипты никуда не устанавливаются. Если вы использовали их, замените свои экземпляры новыми копиями либо внесите те же коррективы в свои вручную изменённые скрипты. Если владельцем существующего файла $PGLOG является root, его нужно удалить или переименовать прежде чем перезапускать сервер с помощью исправленного скрипта. (CVE-2017-12172)

- Исправление вычисления сводных данных индекса BRIN для корректной работы при одновременном расширении таблицы (Альваро Эррера)

  Ранее в условиях гонки некоторые строки таблицы могли пропадать из индекса. Для устранения последствий предыдущих проявлений этой проблемы может потребоваться перестроить существующие индексы BRIN.

- Предупреждение возможных сбоев при параллельных изменениях индекса BRIN (Том Лейн)

  В определённых условиях гонки могли возникать ошибки «invalid index offnum» (неверный номер смещения в индексе) или «inconsistent range map» (несогласованность в карте диапазонов).

- Предотвращение в ходе логической репликации присваивания NULL нереплицируемым столбцам при репликации UPDATE (Петр Желинек)

- Исправление механизма логической репликации, чтобы триггеры BEFORE ROW DELETE вызывались в должное время (Масахико Савада)

  Ранее этого не происходило, если в таблице также имелся триггер BEFORE ROW UPDATE.

- Устранение сбоя при вызове логического декодирования из функции, использующей SPI, в частности, из любой функции на одном из языков PL (Том Лейн)

- Игнорирование CTE (общих табличных выражений) при обращении к целевой таблице для выполнения INSERT/UPDATE/DELETE и недопущение сопоставления имён целевых таблиц, заданных с указанием схемы, с именами переходных таблиц в триггерах (Томас Манро)

  В результате для CTE, связанных с командами DML, восстановлено поведение, имевшее место до версии 10.

- Уход от вычисления выражений в аргументах агрегатной функции для строк, не удовлетворяющих условию FILTER (Том Лейн)

  Тем самым восстанавливается поведение, наблюдавшееся до 10 версии (и описанное в стандарте SQL).

- Исправление некорректных результатов запросов, содержащих в нескольких столбцах GROUPING SETS одну и ту же простую переменную (Том Лейн)

- Устранение утечки памяти на время выполнения запроса при вычислении функции, возвращающей множество, в целевом списке `SELECT` (Том Лейн)
- Реализована возможность параллельного выполнения подготовленных операторов с общими планами (Амит Капила, Кунтал Гхош)
- Корректировка неправильных решений по распараллеливанию для вложенных запросов (Амит Капила, Кунтал Гхош)
- Устранение сбоев в обработке параллельных запросов, возможных при удалении недавно использованной роли (Амит Капила)
- Устранение сбоя при параллельном сканировании битовой карты с узлом плана BitmapAnd, расположенным под узлом BitmapOr (Дилип Кумар)
- Исправление в функциях `json_build_array()`, `json_build_object()` и их аналогах для `jsonb` обработки явно заданных аргументов `VARIADIC` (Микаэль Пакье)
- Исправление логики «частей работы» автоочистки для предупреждения возможных сбоев и незаметной потери частей (Альваро Эррера)
- Устранение сбоев в особых случаях при добавлении столбцов в конец представления (Том Лейн)
- Фиксирование правильных зависимостей когда представление или правило содержит узлы выражения `FieldSelect` или `FieldStore` (Том Лейн)

  В отсутствие этих зависимостей команда `DROP` со столбцом или типом данных может выполниться, когда не должна, что вызовет ошибки при последующем использовании представления или правила. Данное исправление не защищает существующие представления/правила, а повлияет только на создаваемые в будущем.
- Правильное определение хешируемости диапазонных типов данных (Том Лейн)

  Планировщик ошибочно полагал, что любой диапазонный тип может хешироваться для использования в соединениях или агрегировании по хешу, но на самом деле он должен проверять, поддерживает ли хеширование подтип диапазона. Это не касается встроенных диапазонных типов, так как все они всё равно хешируемые.
- Игнорирование узлов выражений `RelabelType` должным образом при рассмотрении статистики по функциональным зависимостям (Дэвид Роули)

  Это позволяет, например, правильно использовать расширенную статистику по столбцам `varchar`.
- Предотвращение разделения информации переходного состояния между сортирующими агрегатными функциями (Дэвид Роули)

  Вследствие этого разделения возникали сбои во встроенных сортирующих агрегатах и могли также возникать в подобных пользовательских функциях. Начиная с версии 11, предусмотрены средства для надёжной защиты от таких случаев, а в ветвях стабильных выпусков просто отключена оптимизация.
- Недопущение игнорирования значения `idle_in_transaction_session_timeout`, если до этого произошёл тайм-аут оператора (`statement_timeout`) (Лукас Фиттл)
- Устранение маловероятной потери сообщений `NOTIFY` вследствие зацикливания идентификаторов транзакций (Марко Тиккая, Том Лейн)

  Если в сеансе не выполнялись никакие запросы, то есть он просто ждал уведомлений и за это время прошло более 2 миллиардов транзакций, он начинал пропускать уведомления от параллельно фиксируемых транзакций.
- Уменьшение частоты запросов на сброс данных в процессе копирования файлов во избежание проблем с производительностью в macOS, в частности с новой файловой системой APFS (Том Лейн)

- Реализована возможность использовать режим `FREEZE` команды `COPY` на уровне изоляции транзакций `REPEATABLE READ` или выше (Ной Миш)

  Этот сценарий использования был непреднамеренно потерян вследствие предыдущего исправления ошибки.

- Исправление функции `AggGetAggref()`, чтобы возвращались корректные узлы `Aggref` для функций завершения агрегатов с объединёнными промежуточными вычислениями (Том Лейн)
- Исправление нечётких указаний схемы в некоторых новых запросах в pg_dump и psql (Виталий Буровой, Том Лейн, Ной Миш)
- Уход от использования оператора `@>` в запросах psql для команды `\d` (Том Лейн)

  Это предотвращает проблемы, возникающие при установке расширения parray_gin, так как в нём определён конфликтующий оператор.

- Исправление в pg_basebackup сравнения путей табличных пространств посредством приведения путей к канонической форме (Микаэль Пакье)

  Это особенно полезно в Windows.

- Исправление libpq, чтобы для её работы не требовалось существование домашнего каталога пользователя (Том Лейн)

  В версии 10 невозможность найти домашний каталог при попытке прочитать `~/.pgpass` считалась критической ошибкой, но эта ситуация должна обрабатываться так же, как и отсутствие данного файла. И в версии 10, и в ветвях предыдущих выпусков была допущена та же ошибка при чтении `~/.pg_service.conf`, хотя это было менее очевидно, так как данный файл использовался только при указании имени службы.

- В ecpglib исправлена обработка обратной косой черты в строковых константах в зависимости от значения `standard_conforming_strings` (Такаюки Цунакава)
- Игнорирование в ecpglib дробной части при вводе целочисленных значений в режиме совместимости с Informix (Гао Цзэнци, Михаэль Мескес)
- Добавление отсутствующего предварительного требования temp-install для целей типа `check` в Make (Ной Миш)

  Некоторые невыполняемые по умолчанию тестовые процедуры, подобные `make check`, не проверяли актуальность временной инсталляции.

- Обновление данных часовых поясов до версии tzdata 2017c, включающее изменения правил перехода на летнее время на Фиджи, в Намибии, Серверном Кипре, Судане, Тонга и на островах Теркс и Кайкос, плюс корректировку исторических данных для Аляски, Апии, Бирмы, Калькутты, Детройта, Ирландии, Намибии и Паго-Паго.
- В документации для HTML-якорей восстановлен верхний регистр (Питер Эйзентраут)

  Вследствие изменений инструментария в документации 10.0 для внутренних HTML-якорей был изменён регистр строк, что привело к повреждению некоторых внешних ссылок на документацию на нашем сайте. Поэтому было решено вернуться к предыдущему стилю со строками в верхнем регистре.

E.50. Выпуск 10

Дата выпуска: 2017-10-05

E.50.1. Обзор

В число ключевых усовершенствований PostgreSQL 10 входят:

- Логическая репликация по схеме публикации/подписки

- Декларативное секционирование таблиц
- Улучшение распараллеливания запросов
- Значительное увеличение общей производительности
- Более сильная защита паролей с использованием SCRAM-SHA-256
- Улучшенные средства мониторинга и управления

Предыдущие пункты более подробно описаны в следующих разделах.

E.50.2. Миграция на версию 10

Тем, кто хочет мигрировать данные из любой предыдущей версии, необходимо выполнить выгрузку/загрузку данных с помощью pg_dumpall либо использовать pg_upgrade или логическую репликацию. Общую информацию о переходе на более новую основную версию можно найти в Разделе 17.6.

В версии 10 реализован ряд изменений, которые могут повлиять на совместимость с предыдущими выпусками. Примите к сведению следующие несовместимости:

- После обновления с помощью pg_upgrade с любой предыдущей основной версии PostgreSQL необходимо перестроить хеш-индексы (Митхун Сай, Роберт Хаас, Амит Капила)

  Необходимость данного требования продиктована значительным усовершенствованием механизма хеш-индексов. Для облегчения задачи переиндексации pg_upgrade создаст вспомогательный скрипт.

- Переименование каталога с журналом предзаписи из `pg_xlog` в `pg_wal`, а также переименование каталога с информацией о состоянии транзакций из `pg_clog` в `pg_xact` (Микаэль Пакье)

  Старые имена неоднократно вводили пользователей в заблуждение — пользователи думали, что эти каталоги содержат только несущественные файлы журналов, и вручную удаляли файлы журналов предзаписи или состояния транзакций, что приводило к необратимой потере данных. Эти переименования призваны предотвратить такие ошибки в будущем.

- Замена в именах функций SQL, утилит и параметров всех упоминаний «xlog» на «wal» (Роберт Хаас)

  Например, имя `pg_switch_xlog()` поменялось на `pg_switch_wal()`, pg_receivexlog — на pg_receivewal, а `--xlogdir` — на `--waldir`. Это было сделано вместе с переименованием каталога `pg_xlog` для согласованности; и вообще термин «xlog» теперь нигде не может встретиться пользователю.

- Замена в связанных с WAL функциях и представлениях `location` на `lsn` (Дэвид Роули)

  Ранее имело место несогласованное употребление обоих терминов.

- Изменение реализации функций, возвращающих множества, в списке `SELECT` (Андрес Фройнд)

  Функции, возвращающие множества, теперь вычисляются до вычисления скалярных выражений в списке `SELECT`, практически так же, как если бы они были помещены в предложение `LATERAL FROM`. Это даёт более понятное поведение в случаях, когда присутствуют несколько таких функций. Если они возвращают разное количество строк, все результаты дополняются до наибольшего количества строк значениями NULL. Ранее результаты обрабатывались в цикле, пока все функции не завершались, и в результате получалось количество строк, равное наименьшему общему кратному периодов функций. Кроме того, функции, возвращающие множества, теперь нельзя использовать в конструкциях `CASE` и `COALESCE`. За дополнительными сведениями обратитесь к Подразделу 35.4.8.

- Использование стандартного синтаксиса конструктора строки в `UPDATE ... SET (`*`список_столбцов`*`) = `*`конструктор_строки`* (Том Лейн)

Теперь *конструктор_строки* может начинаться со слова ROW; ранее его надо было опускать. Если в *списке_столбцов* фигурирует имя только одного столбца, *конструктор_строки* должен использовать ключевое слово ROW, так как иначе он не будет корректным конструктором строки, а будет восприниматься как выражение в скобках. Также запись *имя_таблицы*.* в *конструкторе_строки* теперь разворачивается в набор столбцов, как и в других случаях использования *конструктора_строки*.

- Когда ALTER TABLE ... ADD PRIMARY KEY помечает столбцы как NOT NULL, это изменение теперь распространяется также на дочерние таблицы в иерархии наследования (Микаэль Пакье)
- Предотвращение срабатывания триггеров уровня оператора более одного раза для одного оператора (Том Лейн)

  В случаях, когда пишущие CTE изменяли одну таблицу, изменяемую внешним оператором либо другими пишущими CTE, триггеры BEFORE STATEMENT и AFTER STATEMENT вызывались неоднократно. Также, если в таблице были определены триггеры уровня оператора, в результате действий для обеспечения целостности внешнего ключа (например, ON DELETE CASCADE), они могли вызываться несколько раз для одного внешнего SQL-оператора. Это поведение противоречит стандарту SQL и было исправлено.
- Перемещение полей метаданных последовательностей в новый системный каталог pg_sequence (Питер Эйзентраут)

  Отношение последовательности содержит теперь только поля, которые могут быть изменены функцией nextval(), то есть last_value, log_cnt и is_called. Другие свойства последовательности, такие как начальное значение и шаг увеличения, сохраняются в соответствующей строке в каталоге pg_sequence. Действие ALTER SEQUENCE теперь полностью транзакционное и, как следствие, блокирует последовательность до фиксации транзакции. Функции nextval() и setval() остаются нетранзакционными.

  Основная несовместимость, привнесённая этим изменением, состоит в том, что из отношения последовательности теперь можно прочитать только три поля, перечисленные выше. Чтобы получить другие свойства последовательности, приложения должны обратиться к pg_sequence. Также для этого можно использовать новое системное представление pg_sequences; оно выдаёт столбцы с именами, более подходящими для существующего кода.

  Кроме того, последовательности, созданные для столбцов SERIAL, теперь генерируют 32-битные значения, тогда как предыдущие версии генерировали 64-битные. Это отличие никак не проявляется, если значения просто сохраняются в столбце.

  Переработан и вывод команды psql \d для последовательностей.
- Передача по умолчанию программой pg_basebackup данных WAL, необходимых для восстановления резервной копии (Магнус Хагандер)

  При этом в pg_basebackup подразумеваемое значение -X/--wal-method меняется на stream. Для воспроизведения старого поведения был добавлен вариант значения none. Параметр pg_basebackup -x был удалён (используйте вместо него -X fetch).
- Изменение записей логической репликации в pg_hba.conf (Питер Эйзентраут)

  В предыдущих версиях для соединения логической репликации требовалось указать replication в столбце базы данных. С этого выпуска логической репликации соответствует обычная запись с именем базы данных или ключевым словом, например, all. Для физической репликации по-прежнему используется ключевое слово replication. Так как встроенная логическая репликация появилась только в этом выпуске, это изменение может затронуть только пользователей сторонних средств логической репликации.
- Все действия в pg_ctl по умолчанию должны ожидать завершения операции (Питер Эйзентраут)

Ранее некоторые действия pg_ctl не ждали завершения и для ожидания требовалось использовать ключ `-w`.

- Изменение значения по умолчанию серверного параметра log_directory с `pg_log` на `log` (Андреас Карлссон)
- Добавление параметра конфигурации ssl_dh_params_file для указания имени файла с нестандартными параметрами OpenSSL DH (Хейкки Линнакангас)

  Тем самым заменяется жёстко заданное и недокументированное имя файла `dh1024.pem`. Заметьте, что файл `dh1024.pem` по умолчанию больше не обрабатывается; вы должны задать этот параметр, если хотите использовать свои параметры DH.
- Увеличение размера стандартных параметров DH, используемых для эфемерных DH-шифров OpenSSL, до 2048 бит (Хейкки Линнакангас)

  Размер предопределённых в коде параметров DH был увеличен с 1024 до 2048 бит, так что обмен ключами DH стал более устойчивым к подбору шифра. Однако некоторые старые реализации SSL, в частности некоторые ревизии Java Runtime Environment версии 6, не принимают параметры DH длиннее 1024 бит и, таким образом, не смогут подключиться через SSL. Если вам необходимо поддерживать такие старые клиенты, вы можете использовать нестандартные параметры DH размером 1024 бита вместо предопределённых параметров по умолчанию. См. ssl_dh_params_file.
- Ликвидация возможности хранения незашифрованных паролей на сервере (Хейкки Линнакангас)

  Серверный параметр password_encryption больше не поддерживает значения `off` и `plain`. Вариант `UNENCRYPTED` также теперь не поддерживается в командах `CREATE/ALTER USER ... PASSWORD`. Аналогично был удалён ключ `--unencrypted` команды createuser. Незашифрованные пароли, перенесённые из старых версий, в этой версии будут храниться в зашифрованном виде. Значением по умолчанию параметра `password_encryption` остаётся `md5`.
- Добавление серверных параметров min_parallel_table_scan_size и min_parallel_index_scan_size для управления параллельными запросами (Амит Капила, Роберт Хаас)

  Они заменяют переменную `min_parallel_relation_size`, которая была слишком общей.
- Не заключённый в кавычки текст в shared_preload_libraries и связанных серверных параметрах не должен переводиться в нижний регистр (Куэль Чжо)

  Эти параметры на самом деле представляют собой списки имён файлов, но ранее они воспринимались как списки SQL-идентификаторов и обрабатывались по другим правилам.
- Удаление серверного параметра `sql_inheritance` (Роберт Хаас)

  При выключенном значении этого параметра запросы, обращающиеся к родительским таблицам, перестают учитывать дочерние. Однако стандарт SQL требует, чтобы они учитывались, и это поведение было принято по умолчанию в PostgreSQL 7.1.
- Реализация возможности передавать многомерные массивы в функции на PL/Python и возвращать вложенные списки Python (Алексей Грищенко, Дэйв Крамер, Хейкки Линнакангас)

  Это изменение потребовало нарушить обратную совместимость в части обработки массивов составных типов в PL/Python. Раньше можно было вернуть массив составных типов как `[[col1, col2], [col1, col2]]`; но теперь это будет интерпретироваться как двухмерный массив. Составные типы в массивах для однозначности должны возвращаться теперь в виде кортежей Python, а не в виде списков. То есть предыдущую запись нужно заменить на `[(col1, col2), (col1, col2)]`.
- Удаление механизма автозагрузки «модулей» PL/Tcl (Том Лейн)

На замену этой функциональности пришли новые серверные параметры pltcl.start_proc и pltclu.start_proc, которые проще в использовании и дают возможности, подобные имеющимся в других языках программирования.

- Ликвидация поддержки выгрузки данных с серверов до 8.0 утилитами pg_dump/pg_dumpall (Том Лейн)

  Пользователям, которым нужно выгрузить данные с серверов до 8.0, остаётся использовать dump из PostgreSQL версии 9.6 или старее. Выгруженные этими версиями данные должны успешно загружаться на новых серверах.

- Ликвидация поддержки хранения даты/времени и интервалов в виде чисел с плавающей точкой (Том Лейн)

  Параметр configure `--disable-integer-datetimes` был удалён. Использование для таких значений чисел с плавающей точкой не давало значимых преимуществ и не было вариантом по умолчанию, начиная с PostgreSQL 8.3.

- Ликвидация поддержки сервером клиент-серверного протокола версии 1.0 (Том Лейн)

  Этот протокол не поддерживался клиентами со времён PostgreSQL 6.3.

- Удаление модуля `contrib/tsearch2` (Роберт Хаас)

  Этот модуль обеспечивал совместимость с версией полнотекстового поиска, которая поставлялась с серверами PostgreSQL до версии 8.3.

- Ликвидация приложений командной строки createlang и droplang (Питер Эйзентраут)

  Они перешли в разряд устаревших в PostgreSQL 9.1. Вместо них можно непосредственно использовать команды `CREATE EXTENSION` и `DROP EXTENSION`.

- Ликвидация поддержки вызовов функций версии 0 (Андрес Фройнд)

  Расширения, предоставляющие функции, реализованные на C, теперь должны использовать соглашение о вызовах версии 1. Версия 0 считалась устаревшей с 2001 года.

E.50.3. Изменения

Ниже вы найдёте подробный список изменений, произошедших между предыдущим основным выпуском и PostgreSQL 10.

E.50.3.1. Сервер

E.50.3.1.1. Параллельное выполнение запросов

- Поддержка параллельного сканирования индексов типа B-дерево (Рахила Сьед, Амит Капила, Роберт Хаас, Рафия Сабих)

  Благодаря этому поиск в страницах индекса-B-дерева могут производить параллельно несколько рабочих процессов.

- Поддержка параллельного сканирования кучи по битовой карте (Дилип Кумар)

  Это позволяет распределить одно сканирование индекса между параллельными исполнителями, обрабатывающими разные области кучи.

- Реализована возможность выполнения соединения слиянием в параллельном режиме (Дилип Кумар)

- Реализована возможность выполнения несвязанных запросов в параллельном режиме (Амит Капила)

- Улучшение возможности параллельных процессов возвращать ранее отсортированные данные (Рушаб Латиа)

- Расширение использования параллельных запросов в функциях процедурных языков (Роберт Хаас, Рафия Сабих)
- Добавление серверного параметра max_parallel_workers для ограничения числа рабочих процессов, которые могут использоваться для распараллеливания запросов (Жюльен Руо)

  Значение этого параметра можно сделать меньше max_worker_processes, чтобы зарезервировать рабочие процессы для других целей, кроме параллельных запросов.
- Включение распараллеливания по умолчанию (значение по умолчанию параметра max_parallel_workers_per_gather стало равно `2`).

E.50.3.1.2. Индексы

- Добавление регистрации в журнале предзаписи операций с хеш-индексами (Амит Капила)

  В результате хеш-индексы становятся отказоустойчивыми и пригодными к репликации. Прежнее предостережение относительно их использование удалено.
- Улучшение производительности хеш-индекса (Амит Капила, Митхун Сай, Ашутош Шарма)
- Добавление поддержки индексов SP-GiST для типов данных `INET` и `CIDR` (Эмре Хасегели)
- Добавление параметра для включения более активного вычисления сводных данных индекса BRIN (Альваро Эррера)

  Новый параметр `CREATE INDEX` позволяет автоматически обновлять сводную запись для предыдущей зоны страниц BRIN при создании новой зоны.
- Добавление функций для удаления и повторного добавления сводных записей BRIN для зон индекса BRIN (Альваро Эррера)

  Новая SQL-функция `brin_summarize_range()` обновляет сводные записи индекса BRIN для определённой зоны, а `brin_desummarize_range()` удаляет их. Это полезно для обновления данных зоны, которая стала меньше в результате действия команд `UPDATE` и `DELETE`.
- Более точный расчёт выгоды от применения сканирования по индексу BRIN (Дэвид Роули, Эмре Хасегели)
- Ускорение операций добавления и изменения записей в GiST благодаря более эффективному использованию пространства индекса (Андрей Бородин)
- Минимизация блокировок страниц при очистке индексов GIN (Андрей Бородин)

E.50.3.1.3. Блокировки

- Сокращение блокировок, необходимых для изменения параметров таблиц (Саймон Риггс, Фабрицио де Ройес Мелло)

  Например, изменить параметр effective_io_concurrency для таблицы теперь можно с более лёгкой блокировкой.
- Реализация регулировки пределов для повышения уровня предикатных блокировок (Дагфинн Ильмари Маннсакер)

  Повышением уровня блокировок теперь можно управлять с помощью двух новых параметров сервера, max_pred_locks_per_relation и max_pred_locks_per_page.

E.50.3.1.4. Оптимизатор

- Добавление статистики по нескольким столбцам, позволяющей вычислять коэффициент корреляции и число различных значений (Томаш Вондра, Дэвид Роули, Альваро Эррера)

  Появились новые команды: `CREATE STATISTICS`, `ALTER STATISTICS` и `DROP STATISTICS`. Это полезно для оценки использования памяти запросом и для консолидации статистики по отдельным столбцам.

- Увеличение производительности запросов, сталкивающихся с ограничениями защиты на уровне строк (Том Лейн)

  Теперь оптимизатор лучше понимает, куда можно поместить условия фильтра RLS, благодаря чему он может строить лучшие планы, при этом гарантируя выполнение условий RLS.

E.50.3.1.5. Общая производительность

- Ускорение агрегатных функций, которые вычисляют сумму с накоплением, используя арифметику типа `numeric`, включая некоторые вариации `SUM()`, `AVG()` и `STDDEV()` (Хейкки Линнакангас)
- Увеличение скорости преобразований кодировок символов с использованием цифровых деревьев (Кётаро Хоригути, Хейкки Линнакангас)
- Уменьшение издержек вычисления выражений при выполнении запросов, а также издержек обращения к узлу плана (Андрес Фройнд)

  Это особенно полезно для запросов, обрабатывающих множество строк.
- Возможность использования агрегирования по хешу при обработке наборов группирования (Эндрю Гирт)
- Использование гарантий уникальности для оптимизации определённых типов соединений (Дэвид Роули)
- Ускорение сортировки типа данных `macaddr` (Брандур Лич)
- Сокращение издержек на отслеживание статистики в сеансах, когда задействуются тысячи отношений (Александр Алексеев)

E.50.3.1.6. Мониторинг

- Возможность явного управления отображением командой `EXPLAIN` времени планирования и выполнения (Ашутош Бапат)

  По умолчанию время планирования и выполнения выводится командой `EXPLAIN ANALYZE` и не выводится в других случаях. Явно управлять этим позволяет новый параметр `SUMMARY` команды `EXPLAIN`.
- Добавлены новые стандартные роли для мониторинга (Дейв Пейдж)

  Новые роли `pg_monitor`, `pg_read_all_settings`, `pg_read_all_stats` и `pg_stat_scan_tables` позволяют упростить конфигурацию разрешений.
- Исправление передачи информации сборщику статистики во время `REFRESH MATERIALIZED VIEW` (Джим Млодженски)

E.50.3.1.6.1. Ведение журнала

- Изменение префикса log_line_prefix, чтобы по умолчанию все строки журнала postmaster содержали текущее время (с миллисекундами) и идентификатор процесса (Кристоф Берг)

  Ранее префикс по умолчанию был пустым.
- Добавление функций для получения содержимого каталогов журнала сообщений и WAL (Дейв Пейдж)

  Новые функции называются соответственно `pg_ls_logdir()` и `pg_ls_waldir()` и могут выполняться не только суперпользователями (при наличии достаточных разрешений).
- Добавление функции `pg_current_logfile()` для чтения имён текущих файлов, в которые сборщик сообщений выводит потоки stderr и csvlog (Жиль Даролд)
- Вывод в журнал сервера адреса и номера порта для каждого принимающего соединения сокета при запуске postmaster (Том Лейн)

Также добавление в сообщение об ошибке привязки к определённому сокету адреса этого сокета.

- Устранение лишних сообщений о запуске и остановке подпроцессов запускающего процесса (Том Лейн)

  Теперь уровень этих сообщений понижен до `DEBUG1`.

- Уменьшение уровня важности сообщений для уровней отладки, контролируемых параметром log_min_messages (Роберт Хаас)

  Это затрагивает также сообщения отладочных уровней client_min_messages.

E.50.3.1.6.2. Представление `pg_stat_activity`

- Добавление в `pg_stat_activity` информации о состоянии низкоуровневых ожиданий (Микаэль Пакье, Роберт Хаас)

  Это позволяет отслеживать множество событий ожидания на низком уровне, включая ожидания защёлок, записи/чтения/сброса файлов, чтения/записи со стороны клиента и синхронной репликации.

- Отображение в `pg_stat_activity` вспомогательных и фоновых рабочих процессов, а также процессов-передатчиков WAL (Кунтал Гхош, Микаэль Пакье)

  Это упрощает мониторинг. Тип процесса обозначается в новом столбце `backend_type`.

- В `pg_stat_activity` реализовано отображение SQL-запросов, выполняемых параллельными исполнителями (Рафия Сабих)

- Переименование в `pg_stat_activity.wait_event_type` значений `LWLockTranche` и `LWLockNamed` в `LWLock` (Роберт Хаас)

  Это делает вывод более согласованным.

E.50.3.1.7. Аутентификация

- Добавление поддержки SCRAM-SHA-256 для проверки и хранения паролей (Микаэль Пакье, Хейкки Линнакангас)

  Это обеспечивает лучшую безопасность по сравнению с существующим методом `md5`.

- Изменение типа серверного параметра password_encryption с `boolean` на `enum` (Микаэль Пакье)

  Это потребовалось для поддержки различных вариантов хеширования паролей.

- Добавление представления `pg_hba_file_rules` для просмотра содержимого `pg_hba.conf` (Харибабу Комми)

  Это представление показывает содержимое файла, а не действующие в данный момент параметры.

- Поддержка нескольких серверов RADIUS (Магнус Хагандер)

  Все связанные с RADIUS параметры теперь стали множественными и принимают список серверов через запятую.

E.50.3.1.8. Конфигурация сервера

- Возможность обновления конфигурации SSL при перезагрузке конфигурации (Андреас Карлссон, Том Лейн)

  Это позволяет переконфигурировать SSL без перезапуска сервера, используя `pg_ctl reload`, `SELECT pg_reload_conf()` или отправив сигнал `SIGHUP`. Однако перезагрузка конфигурации

SSL не сработает, если для использования SSL-ключа сервера требуется пароль, так как никакой возможности запросить его нет. В этом случае главный процесс (postmaster) продолжит использовать изначальную конфигурацию, пока не будет перезапущен.

- Устранение практического предела для максимального значения bgwriter_lru_maxpages (Джим Нэсби)

E.50.3.1.9. Надёжность

- После создания или удаления файлов следует выполнять fsync для каталога их содержащего (Микаэль Пакье)

  Это сокращает риск потери данных при отключении питания.

E.50.3.1.9.1. Журнал предзаписи (WAL)

- Предотвращение ненужных контрольных точек и архивации WAL в простаивающих системах (Микаэль Пакье)
- Добавление серверного параметра wal_consistency_checking для внесения в WAL информации, позволяющей проверять целостность на ведомом сервере (Кунтал Гхош, Роберт Хаас)

  В случае выявления любого нарушения при проверке целостности на ведомом сервере выдаётся критическая ошибка.
- Увеличение максимально допустимого размера сегмента WAL до одного гигабайта (Бина Эмерсон)

  Увеличивая размер сегментов WAL, можно сократить частоту вызова archive_command и уменьшить число файлов WAL.

E.50.3.2. Репликация и восстановление

- Реализация возможности логически реплицировать таблицы на подчинённые серверы (Петр Желинек)

  Логическая репликация даёт большую гибкость, чем физическая; в том числе позволяет организовывать репликацию между разными основными версиями PostgreSQL, а также избирательную репликацию.
- Возможность ожидания подтверждения фиксации от ведомых серверов вне зависимости от их порядка в списке synchronous_standby_names (Масахико Савада)

  Ранее сервер всегда ожидал ответа от активных ведомых серверов, стоящих первыми в списке `synchronous_standby_names`. Новое ключевое слово `ANY` в `synchronous_standby_names` позволяет выбрать ожидание любого числа серверов, вне зависимости от их порядка. Это называется фиксацией на основе кворума.
- Упрощение изменений конфигурации, которые необходимо внести для организации потокового копирования и репликации (Магнус Хагандер, Дан Мин Хьюнг)

  В частности, были изменены значения по умолчанию для параметров wal_level, max_wal_senders, max_replication_slots и hot_standby, чтобы они были пригодны для такого использования в исходном состоянии.
- По умолчанию репликация разрешается для локальных подключений в `pg_hba.conf` (Микаэль Пакье)

  Ранее строки для соединений репликации в `pg_hba.conf` по умолчанию были закомментированы. Это особенно полезно для pg_basebackup.
- Добавление в `pg_stat_replication` столбцов с информацией о задержках репликации (Томас Манро)

  Новые столбцы называются `write_lag`, `flush_lag` и `replay_lag`.

- Возможность указания точки остановки восстановления по LSN (последовательному номеру в журнале) в `recovery.conf` (Микаэль Пакье)

  Ранее точку остановки можно было задать только по времени или идентификатору транзакции.

- Предоставление пользователям возможности отключить в функции `pg_stop_backup()` ожидание архивации всех файлов WAL (Дэвид Стил)

  Этим поведением управляет необязательный второй аргумент функции `pg_stop_backup()`.

- Возможность создания временных слотов репликации (Петр Желинек)

  Временные слоты автоматически удаляются при завершении сеанса или при ошибке.

- Увеличение производительности воспроизведения в режиме горячего резерва благодаря оптимизации обработки исключительных блокировок (Саймон Риггс, Дэвид Роули)
- Ускорение восстановления при двухфазной фиксации (Стас Кельвич, Никхил Сонтакке, Микаэль Пакье)

E.50.3.3. Запросы

- Добавление функции `XMLTABLE`, преобразующей данные в формате `XML` в набор табличных строк (Павел Стехуле, Альваро Эррера)
- Исправление обработки в регулярных выражениях классов символов для символов с большими кодами, в частности, для символов Unicode больше `U+7FF` (Том Лейн)

  Ранее такие символы никогда не воспринимались как принадлежащие классам, зависящим от локали, например `[[:alpha:]]`.

E.50.3.4. Служебные команды

- Добавление синтаксиса секционирования для автоматического создания ограничений секций и распределения операций добавления и изменения кортежей (Амит Ланготе)

  Этот синтаксис поддерживает секционирование по спискам и по диапазонам.

- Добавление для триггеров `AFTER` переходных таблиц, содержащих изменённые строки (Кевин Гриттнер, Томас Манро)

  Содержимое переходных таблиц доступно в триггерах, написанных на языках программирования на стороне сервера.

- Реализация ограничительных политик защиты на уровне строк (Стивен Фрост)

  Ранее все политики были разрешительными, то есть при выполнении условия любой политики доступ разрешался. Но теперь при невыполнении ограничительной политики доступ будет запрещён. Эти типы политик можно комбинировать вместе.

- При создании ограничения внешнего ключа разрешение `REFERENCES` должно требоваться только для целевой таблицы (Том Лейн)

  Ранее также требовалось разрешение `REFERENCES` в таблице внешнего ключа. Это требование было реализовано из-за недопонимания SQL-стандарта. Так как для создания в таблице ограничения внешнего ключа (или подобного) требуются иметь права владельца этой таблицы, дополнительное требование разрешения `REFERENCES` кажется довольно бессмысленным.

- Реализация прав по умолчанию для схем (Матеус Оливейра)

  Эти права задаются командой `ALTER DEFAULT PRIVILEGES`.

- Добавление команды `CREATE SEQUENCE AS` для создания последовательностей определённого целочисленного типа данных (Питер Эйзентраут)

Это упрощает создание последовательностей с интервалом значений, соответствующим типу базовых столбцов.

- Поддержка команды `COPY` *`представление`* `FROM` *`источник`* для представлений с триггерами `INSTEAD INSERT` (Харибабу Комми)

  Триггеры получают строки данных, которые читает команда `COPY`.

- Допущение указания имени функции без аргументов в командах DDL, если это имя уникально (Питер Эйзентраут)

  Например, теперь допускается указание `DROP FUNCTION` с именем функции без аргументов, если существует только одна функция с таким именем. Такое поведение требуется стандартом SQL.

- Реализация возможности удалять несколько функций, операторов и агрегатов одной командой `DROP` (Питер Эйзентраут)

- Поддержка предложения `IF NOT EXISTS` в командах `CREATE SERVER`, `CREATE USER MAPPING` и `CREATE COLLATION` (Анастасия Лубенникова, Питер Эйзентраут)

- Добавление в вывод `VACUUM VERBOSE` количества пропущенных замороженных страниц и старейшего xmin (Масахико Савада, Саймон Риггс)

  Эта информация также включается в вывод с log_autovacuum_min_duration.

- Ускорение удаления конечных пустых страниц кучи в процессе операции `VACUUM` (Клаудио Фрейре, Альваро Эррера)

E.50.3.5. Типы данных

- Добавление поддержки полнотекстового поиска для типов `JSON` и `JSONB` (Дмитрий Долгов)

  С этими типами данных теперь могут использоваться функции `ts_headline()` и `to_tsvector()`.

- Добавление поддержки MAC-адресов EUI-64 в виде нового типа данных `macaddr8` (Харибабу Комми)

  Этот тип дополняет ранее реализованную поддержку MAC-адресов EUI-48 (тип `macaddr`).

- Добавление столбцов идентификации для назначения числовых значений столбцам при добавлении данных (Питер Эйзентраут)

  Этот механизм подобен столбцам `SERIAL`, но соответствует стандарту SQL.

- Реализация возможности переименовывать значения `ENUM` (Дагфинн Ильмари Маннсакер)

  Для этого используется синтаксис `ALTER TYPE ... RENAME VALUE`.

- Исправление обращения с псевдотипами массивов (`anyarray`) как с массивами в функциях `to_json()` и `to_jsonb()` (Эндрю Дунстан)

  Ранее столбцы, объявленные как `anyarray`, (в частности, столбцы в представлении `pg_stats`) преобразовывались не в массивы, а в строки `JSON`.

- Добавление операторов для умножения и деления значений `money` на значения `int8` (Питер Эйзентраут)

  Ранее в таких случаях значения `int8` приводились к типу `float8`, а затем использовались операторы `money`-и-`float8`. Новое поведение предупреждает возможную потерю точности. Но заметьте, что при делении `money` на `int8` теперь дробная часть отбрасывается, как и в других случаях целочисленного деления, тогда как ранее происходило округление.

- Проверка переполнения в функции ввода типа `money` (Питер Эйзентраут)

E.50.3.6. Функции

- Добавление упрощённой функции `regexp_match()` (Эмре Хасегели)

  Эта функция подобна `regexp_matches()`, но возвращает только результаты первого вхождения, поэтому для её результата не требуется множество, так что её удобнее использовать в простых случаях.

- Новая версия оператора удаления для `jsonb`, принимающая массив ключей для удаления (Магнус Хагандер)

- Реализация рекурсивной обработки объектов и массивов JSON в функции `json_populate_record()` и родственных ей (Никита Глухов)

  Благодаря этому изменению массивы JSON корректно преобразуются в поля-массивы в целевом типе SQL, а объекты JSON — в поля, представляющие собой составные типы. Ранее в таких случаях происходила ошибка при попытке передать строковое представление JSON-значения функциям `array_in()` и `record_in()`, так как синтаксис полученной строки не соответствовал ожидаемому этими функциями.

- Добавление функции `txid_current_if_assigned()`, возвращающей идентификатор текущей транзакции или `NULL`, если транзакции не присвоен идентификатор (Крейг Рингер)

  Этим данная функция отличается от `txid_current()`, которая всегда возвращает идентификатор транзакции (назначая его, если он отсутствует). Данную функцию, в отличие от неё, можно выполнять на ведомых серверах.

- Добавление функции `txid_status()`, проверяющей, была ли зафиксирована транзакция (Крейг Рингер)

  Это позволяет проверить после неожиданного обрыва соединения, была ли зафиксирована предыдущая транзакция, если вы просто не успели получить уведомление об этом.

- Добавление функции `make_date()`, интерпретирующей отрицательные числа, как годы до нашей эры (Альваро Эррера)

- Функции `to_timestamp()` и `to_date()` не должны принимать поля вне допустимых пределов (Артур Закиров)

  Например, ранее вызов `to_date('2009-06-40','YYYY-MM-DD')` выполнялся успешно и возвращалась дата `2009-07-10`. Теперь с такими аргументами будет выдаваться ошибка.

E.50.3.7. Языки программирования на стороне сервера

- Возможность вызывать функции `cursor()` и `execute()` в PL/Python как методы их аргумента — объекта plan (Питер Эйзентраут)

  Это позволяет придерживаться более объектно-ориентированного стиля программирования.

- Реализована возможность получать значения оператора `GET DIAGNOSTICS` в PL/pgSQL в элементах массива (Том Лейн)

  Ранее синтаксическое ограничение не допускало использования в качестве целевой переменной элемента массива.

E.50.3.7.1. PL/Tcl

- Возможность возвращать составные типы и наборы данных из функций PL/Tcl (Карл Лехенбауэр)

- Добавление команды subtransaction в PL/Tcl (Виктор Вагнер)

Это позволяет обработать ошибку в запросах PL/Tcl, не прерывая всю функцию.

- Добавление серверных параметров pltcl.start_proc и pltclu.start_proc для указания функций инициализации, которые будут вызываться при запуске PL/Tcl (Том Лейн)

E.50.3.8. Клиентские интерфейсы

- Реализована возможность указания нескольких адресов или имён серверов в строках подключения libpq и в строках URI (Роберт Хаас, Хейкки Линнакангас)

  Функции libpq будут подключаться к первому работоспособному узлу из этого списка.

- В строках подключения libpq и URI добавлена возможность затребовать сервер для чтения/записи, то есть ведущий, а не ведомый сервер (Виктор Вагнер, Митхун Сай)

  Это полезно при указании нескольких имён узлов. Соответствующий параметр подключения libpq — `target_session_attrs`.

- Реализована возможность задавать в параметрах подключения libpq имя файла паролей (Джулиан Маркворт)

  Ранее его можно было задать только в переменной окружения.

- Добавлена функция `PQencryptPasswordConn()`, позволяющая создавать больше видов зашифрованных паролей на стороне клиента (Микаэль Пакье, Хейкки Линнакангас)

  Ранее функцией `PQencryptPassword()` можно было создавать только пароли, зашифрованные `MD5`. Новая функция может также создавать и пароли, зашифрованные методом `SCRAM-SHA-256`.

- Изменение версии препроцессора ecpg с 4.12 на 10 (Том Лейн)

  Впредь версия ecpg будет соответствовать номеру версии дистрибутива PostgreSQL.

E.50.3.9. Клиентские приложения

- Добавлена поддержка условных ветвлений в psql (Кори Хинкер)

  В psql появились метакоманды `\if`, `\elif`, `\else` и `\endif`. Это полезно в первую очередь для скриптов.

- Добавление в psql метакоманды `\gx` для выполнения запроса (запуска команды `\g`) в расширенном режиме (`\x`) (Кристоф Берг)

- Расширение переменных psql внутри строк, заключённых в обратные апострофы (Том Лейн)

  Это особенно полезно в новых командах условного ветвления psql.

- Недопущение установки некорректных значений для специальных переменных psql (Даниэль Верите, Том Лейн)

  Ранее, если какой-либо специальной переменной psql присваивалось некорректное значение, оно просто игнорировалось и применялось поведение по умолчанию. Теперь `\set` со специальной переменной выдаст ошибку при недопустимом новом значении. В виде особого исключения `\set` с пустым или без нового значения по-прежнему действует как присваивание переменной значения `on`; но теперь в переменной действительно оказывается это значение, а не пустая строка. Команда `\unset` со специальной переменной теперь действительно сбрасывает переменную так, что она получает значение по умолчанию, то есть то значение, которая она имела при запуске. В итоге управляющая переменная теперь всегда имеет отображаемое значение, отражающее, что фактически делает psql.

- Добавление переменных, показывающих версию сервера и psql (Фабьен Коэльо)

- Добавление в команды `\d` (показать отношение) и `\dD` (показать домен) вывода правила сортировки, допустимости NULL и свойств по умолчанию в отдельных столбцах (Питер Эйзентраут)

  Ранее вся эта информация выводилась в одном столбце «Модификаторы».

- Унификация поведения команд `\d` в случаях, когда запрошенный объект не найден (Даниэль Густафссон)

  Теперь все они выводят сообщения об этом в stderr, а не в stdout, и текст сообщения стал более единообразным.

- Усовершенствование дополнения табуляцией в psql (Джефф Джейнс, Иэн Барвик, Андреас Карлссон, Сероп Саркуни, Томас Манро, Кевин Гриттнер. Дагфинн Ильмари Маннсакер)

- Добавление в pgbench параметра `--log-prefix` для установки префикса файла журнала (Масахико Савада)

- Возможность разбивать метакоманды pgbench на несколько строк (Фабьен Коэльо)

  Теперь метакоманду можно продолжить на следующей строке, введя обратную косую черту перед переводом строки.

- Устранение ограничения на положение параметра `-M` по отношению к другим параметрам командной строки (Том Лейн)

E.50.3.10. Серверные приложения

- Добавление в pg_receivewal параметра `-Z`/`--compress` для включения сжатия (Микаэль Пакье)

- Добавление в pg_recvlogical параметра `--endpos` для указания конечной позиции (Крейг Рингер)

  Этот параметр дополняет существовавший ранее параметр `--startpos`.

- Переименование параметров initdb `--noclean` и `--nosync` в `--no-clean` и `--no-sync` (Вик Фиринг, Питер Эйзентраут)

  Старое написание по-прежнему поддерживается.

E.50.3.10.1. pg_dump, pg_dumpall, pg_restore

- В pg_restore реализована возможность исключать схемы (Михаэль Банк)

  Для этого добавлен новый ключ `-N`/`--exclude-schema`.

- В pg_dump добавлен параметр `--no-blobs` (Гийом Леларж)

  Этот ключ отключает выгрузку больших объектов.

- В pg_dumpall добавлен ключ `--no-role-passwords` для отказа от чтения паролей ролей (Робинс Таракан, Саймон Риггс)

  Это позволяет использовать pg_dumpall не только суперпользователям; без этого ключа pg_dumpall пытается прочитать пароли, а обычным пользователям это не разрешается.

- Поддержка использования синхронных снимков при выгрузке данных с ведомого сервера (Петр Желинек)

- Выполнение `fsync()` (синхронизации с ФС) для файлов, записываемых утилитами pg_dump и pg_dumpall (Микаэль Пакье)

  Это позволяет более уверенно гарантировать, что выгруженные файлы безопасно сохранены на диске, до завершения работы программы. Это поведение можно отключить новым ключом `--no-sync`.

- В pg_basebackup реализована возможность передавать журнал предзаписи в режиме tar (Магнус Хагандер)

  Содержимое WAL будет храниться в отдельном от основной копии файле tar.

- Использование в pg_basebackup временных слотов репликации (Магнус Хагандер)

  Временные слоты репликации будут задействоваться по умолчанию, когда pg_basebackup использует передачу WAL с параметрами по умолчанию.

- Более аккуратное выполнение синхронизации с ФС везде, где это требуется в pg_basebackup и pg_receivewal (Микаэль Пакье)

- Добавление в pg_basebackup ключа `--no-sync` для отключения синхронизации с ФС (Микаэль Пакье)

- Улучшение обработки в pg_basebackup пропускаемых каталогов (Дэвид Стил)

- Добавление параметра ожидания для операции повышения роли в pg_ctl (Питер Эйзентраут)

- Добавление длинных ключей для работы pg_ctl с ожиданием (`--wait`) и без ожидания (`--no-wait`) (Вик Фиринг)

- Добавление длинного ключа для параметров, которые pg_ctl передаёт серверу (`--options`) (Питер Эйзентраут)

- В ожидании готовности сервера команда `pg_ctl start --wait` должна наблюдать за `postmaster.pid`, а не пытаться установить подключение (Том Лейн)

  Процесс postmaster теперь может сигнализировать о своей готовности принимать подключения в файле `postmaster.pid`, и pg_ctl теперь проверяет этот файл, чтобы определить, что сервер запущен. Этот метод эффективнее и надёжнее старого, а кроме того он избавляет от сообщений о неудавшихся попытках подключения при запуске.

- Уменьшение времени реакции pg_ctl в ожидании запуска/остановки процесса postmaster (Том Лейн)

  Теперь pg_ctl проверяет изменения состояния postmaster не один (как раньше), а десять раз в секунду.

- Возврат ненулевого кода состояния при выходе pg_ctl, если ожидаемая операция не завершилась за отведённое время (Питер Эйзентраут)

  Теперь операции `start` и `promote` возвращают в таких случаях код состояния 1, а не 0. Операция `stop` делала это всегда.

E.50.3.11. Исходный код

- Переход на нумерацию версий с двумя компонентами (Питер Эйзентраут, Том Лейн)

  Номера выпусков теперь состоят из двух частей (например, `10.1`), а не из трёх (например, `9.6.3`). Основные версии теперь будут увеличивать только первое число, а корректирующие выпуски — только второе. В названии ветвей выпусков будет включаться только одно число (например, `10` вместо `9.6`). Это изменение произведено для того, чтобы пользователям было понятнее, что есть основная, а что — дополнительная версия PostgreSQL.

- Улучшение поведения pgindent (Пётр Стефаняк, Том Лейн)

  Мы перешли на новую версию pg_bsd_indent, включающую последние усовершенствования из проекта FreeBSD. В результате устранено множество мелких ошибок, которые приводили к странным решениям относительно форматирования кода C. Одно из самых заметных изменений — строки в скобках (например, вызов функции, записанный в нескольких строках)

теперь единообразно выравниваются по открывающей скобке, даже если в результате код будет выходить за правую границу.

- Возможность использования библиотеки ICU для поддержки правил сортировки (Питер Эйзентраут)

  Библиотека ICU реализует версионирование, что позволяет выявлять изменения правил сортировки от версии к версии. Она подключается параметром configure `--with-icu`. По умолчанию для сортировки по-прежнему используется встроенная библиотека операционной системы.

- Автоматическое добавление для всех функций `PG_FUNCTION_INFO_V1` пометки `DLLEXPORT` в Windows (Лауренц Альбе)

  Если сторонний код использует объявления функций с характеристикой `extern`, они также должны иметь пометки `DLLEXPORT` в этих объявлениях.

- Ликвидация ставших ненужными SPI-функций `SPI_push()`, `SPI_pop()`, `SPI_push_conditional()`, `SPI_pop_conditional()` и `SPI_restore_connection()` (Том Лейн)

  Теперь их функциональность реализуется автоматически. Сейчас остались пустые макросы с такими именами (чтобы не требовалось немедленно обновлять внешние модули), но в конце концов их вызовы должны быть удалены.

  Побочным эффектом этого изменения стало то, что `SPI_palloc()` и родственные функции теперь требуют активного SPI-подключения; их действие не сводится к простому `palloc()`, если его нет. Предыдущее такое поведение оказалось не очень полезным и было чревато неожиданными утечками памяти.

- Возможность динамического выделения разделяемой памяти (Томас Манро, Роберт Хаас)
- Добавление механизма выделения памяти, подобного slab, для эффективного выделения памяти фиксированного размера (Томаш Вондра)
- Использование семафоров POSIX вместо SysV в Linux и FreeBSD (Том Лейн)

  Это снимает некоторые присущие платформе лимиты на использование семафоров SysV.

- Улучшение поддержки 64-битных атомарных операций (Андрес Фройнд)
- Использование 64-битных атомарных операций на платформе ARM64 (Роман Шапошник)
- Переход к использованию функции `clock_gettime()`, если она доступна, для замеров длительности (Том Лейн)

  Если `clock_gettime()` недоступна, по-прежнему будет использоваться `gettimeofday()`.

- Добавление более надёжных генераторов случайных чисел для криптографического использования (Магнус Хагандер, Микаэль Пакье, Хейкки Линнакангас)

  Если надёжный генератор случайных чисел не обнаруживается, процедура configure прервётся, если только дополнительно не передать ключ `--disable-strong-random`. Однако с этим ключом функции pgcrypto, которым требуется надёжный генератор случайных чисел, будут отключены.

- В функции `WaitLatchOrSocket()` налажено ожидание подключения сокета в Windows (Андрес Фройнд)
- Функции `tupconvert.c` более не преобразуют кортежи только для того, чтобы включить в них другой OID составного типа (Ашутош Бапат, Том Лейн)

  В большинстве точек вызова этот OID составного типа не нужен; но если результирующий кортеж будет использоваться как составное значение Datum, требуются дополнительные действия, чтобы в нём оказался корректный OID.

- Ликвидация портов SCO и Unixware (Том Лейн)
- Переработка процесса сборки документации (Александр Лахин)
- Использование XSLT для сборки документации PostgreSQL (Питер Эйзентраут)

  Ранее использовались программы Jade, DSSSL и JadeTex.
- Сборка HTML-документации по умолчанию со стилями XSLT (Питер Эйзентраут)

E.50.3.12. Дополнительные модули

- Реализация в file_fdw возможности читать вывод программ, так же как и содержимое файлов (Кори Хинкер, Адам Гомаа)
- В postgres_fdw реализована передача агрегатных функций на удалённый сервер, когда это возможно (Дживан Чок, Ашутош Бапат)

  Это сокращает объём данных, который должен передаваться со стороннего сервера и избавляет запрашивающий сервер от вычисления агрегатной функции.
- postgres_fdw может передавать соединения отношений на сторонний сервер в большем числе случаев (Дэвид Роули, Ашутош Бапат, Эцуро Фудзита)
- Исправление поддержки столбцов `OID` в таблицах postgres_fdw (Эцуро Фудзита)

  Ранее в столбцах `OID` всегда возвращались нули.
- Реализация в btree_gist и btree_gin возможности индексировать типы-перечисления (Эндрю Дунстан)

  Это позволяет использовать перечисления в ограничениях-исключениях.
- Добавление в btree_gist поддержки индексации для типа данных `UUID` (Пол Юнгвирт)
- Добавлено расширение amcheck, которое может проверять корректность индексов-B-деревьев (Питер Гейган)
- Отображение в pg_stat_statements игнорируемых констант как `$N`, а не как `?` (Лукас Фиттл)
- Улучшение в модуле cube обработки кубов с нулевой размерностью (Том Лейн)

  При этом также улучшена работа со значениями `infinite` и `NaN`.
- Сокращение числа блокировок при обращении к представлению pg_buffercache (Иван Картышов)

  Как следствие, его можно более беспрепятственно использовать в производственных средах.
- Добавление в pgstattuple функции `pgstathashindex()` для просмотра статистики хеш-индексов (Ашутош Шарма)
- Возможность использовать `GRANT` для управления доступом к функциям pgstattuple (Стивен Фрост)

  Это позволяет администраторам разрешать вызывать эти функции не только суперпользователям.
- Сокращение числа блокировок в pgstattuple при просмотре хеш-индексов (Амит Капила)
- Добавление в pageinspect функции `page_checksum()` для получения контрольной суммы страницы (Томаш Вондра)
- Добавление в pageinspect функции `bt_page_items()`, которая выводит элементы страницы для переданного образа страницы (Томаш Вондра)
- Добавление поддержки хеш-индексов в pageinspect (Джеспер Педерсен, Ашутош Шарма)

E.50.4. Благодарственный список

Перечисленные ниже (в алфавитном порядке) лица сделали вклад в этот выпуск, разрабатывая, совершенствуя и рецензируя код, принимая правки, проводя тестирование или сообщая о проблемах.

Адам Брайтвелл (Adam Brightwell)
Адам Брюссельбек (Adam Brusselback)
Адам Гомаа (Adam Gomaa)
Адам Сах (Adam Sah)
Адриан Клавер (Adrian Klaver)
Аидан Ван Дик (Aidan Van Dyk)
Александр Алексеев (Aleksander Alekseev)
Александр Коротков (Alexander Korotkov)
Александр Лахин (Alexander Lakhin)
Александр Сосна (Alexander Sosna)
Алексей Баштанов (Alexey Bashtanov)
Алексей Грищенко (Alexey Grishchenko)
Алексей Исайко (Alexey Isayko)
Альваро Эрнандес Тортоза (Álvaro Hernández Tortosa)
Альваро Эррера (Álvaro Herrera)
Амит Капила (Amit Kapila)
Амит Ланготе (Amit Langote)
Амит Хандекар (Amit Khandekar)
Амул Сул (Amul Sul)
Анастасия Лубенникова (Anastasia Lubennikova)
Андреас Джозеф Крог (Andreas Joseph Krogh)
Андреас Зельтенрейх (Andreas Seltenreich)
Андреас Карлссон (Andreas Karlsson)
Андреас Шербаум (Andreas Scherbaum)
Андрей Бородин (Andrey Borodin)
Андрей Лизенко (Andrey Lizenko)
Андрес Фройнд (Andres Freund)
Антонин Хоуска (Antonin Houska)
Антс Аасма (Ants Aasma)
Арсений Шер (Arseny Sher)
Артур Закиров (Artur Zakirov)
Арьен Нинхаус (Arjen Nienhuis)
Атсуши Торикоши (Atsushi Torikoshi)
Ашвин Агравал (Ashwin Agrawal)
Ашутош Бапат (Ashutosh Bapat)
Ашутош Шарма (Ashutosh Sharma)
Аюми Исии (Ayumi Ishii)
Бейзил Бурк (Basil Bourque)
Бен де Грааф (Ben de Graaff)
Бенедикт Грундман (Benedikt Grundmann)
Бернд Хелмле (Bernd Helmle)
Бина Эмерсон (Beena Emerson)
Брандур Лич (Brandur Leach)
Брин Хаган (Breen Hagan)
Бруно Вольф III (Bruno Wolff III)
Брэд Дейонг (Brad DeJong)
Брюс Момджян (Bruce Momjian)
Вайшнави Прабакаран (Vaishnavi Prabakaran)
Венката Балажи Наготи (Venkata Balaji Nagothi)
Вик Фиринг (Vik Fearing)
Вики Вергара (Vicky Vergara)
Виктор Вагнер (Victor Wagner)

Винаяк Покале (Vinayak Pokale)
Вирен Негги (Viren Negi)
Виталий Буровой (Vitaly Burovoy)
Владимир Кунщиков (Vladimir Kunshchikov)
Владимир Русинов (Vladimir Rusinov)
Габриэль Бартолини (Gabriele Bartolini)
Габриэль Рот (Gabrielle Roth)
Гао Цзэнци (Gao Zengqi)
Генри Болерт (Henry Boehlert)
Гердан Сантос (Gerdan Santos)
Гийом Леларж (Guillaume Lelarge)
Грег Аткинс (Greg Atkins)
Грег Бурек (Greg Burek)
Григорий Смолкин (Grigory Smolkin)
Грэхем Даттон (Graham Dutton)
Давид Феттер (David Fetter)
Дагфинн Ильмари Маннсакер (Dagfinn Ilmari Mannsåker)
Дайсукэ Хигути (Daisuke Higuchi)
Дамиан Кирога (Damian Quiroga)
Дан Мин Хыонг (Dang Minh Huong)
Даниель Вестерман (Daniel Westermann)
Данило Глинский (Danylo Hlynskyi)
Даниэле Вараццо (Daniele Varrazzo)
Даниэль Верите (Daniel Vérité)
Даниэль Густафссон (Daniel Gustafsson)
Дарко Прелец (Darko Prelec)
Деврим Гюндюз (Devrim Gündüz)
Дейв Крамер (Dave Cramer)
Дейв Пейдж (Dave Page)
Денис Смирнов (Denis Smirnov)
Дениш Патель (Denish Patel)
Деннис Бьорклунд (Dennis Björklund)
Джанни Чиолли (Gianni Ciolli)
Джаред Уорд (Jarred Ward)
Джастин Муис (Justin Muise)
Джастин Призби (Justin Pryzby)
Джеймс Паркс (James Parks)
Джейсон Ли (Jason Li)
Джейсон О'Доннелл (Jason O'Donnell)
Джейсон Петерсен (Jason Petersen)
Джереми Финцель (Jeremy Finzel)
Джереми Шнайдер (Jeremy Schneider)
Джеспер Педерсен (Jesper Pedersen)
Джефф Девис (Jeff Davis)
Джефф Джейнс (Jeff Janes)
Джефф Дэфо (Jeff Dafoe)
Дживан Ладхе (Jeevan Ladhe)
Дживан Чок (Jeevan Chalke)
Джим Млодженски (Jim Mlodgenski)
Джим Нэсби (Jim Nasby)
Джинью Чжан (Jinyu Zhang)
Джо Конвей (Joe Conway)
Джон Харви (John Harvey)
Джордан Гигов (Jordan Gigov)
Джош Беркус (Josh Berkus)
Джош Сореф (Josh Soref)
Джоэл Джейкобсон (Joel Jacobson)
Джузеппе Брокколо (Giuseppe Broccolo)

Джулиан Маркворт (Julian Markwort)
Джун-сок Ян (Junseok Yang)
Дилип Кумар (Dilip Kumar)
Дилян Палаузов (Dilyan Palauzov)
Дин Рашид (Dean Rasheed)
Дмитрий Долгов (Dmitry Dolgov)
Дмитрий Иванов (Dimitry Ivanov)
Дмитрий Павлов (Dima Pavlov)
Дмитрий Сарафанников (Dmitriy Sarafannikov)
Дмитрий Федин (Dmitry Fedin)
Дон Моррисон (Don Morrison)
Дэвид Джонстон (David Johnston)
Дэвид Кристенсен (David Christensen)
Дэвид Роули (David Rowley)
Дэвид Рэйдер (David Rader)
Дэвид Стил (David Steele)
Дэн Вуд (Dan Wood)
Евгений Казаков (Eugene Kazakov)
Евгений Коньков (Eugen Konkov)
Егор Рогов (Egor Rogov)
Жиль Даролд (Gilles Darold)
Жюльен Руо (Julien Rouhaud)
И Вэнь Вон (Yi Wen Wong)
Иван Картышов (Ivan Kartyshov)
Игорь Корот (Igor Korot)
Ильдус Курбангалиев (Ildus Kurbangaliev)
Иэн Барвик (Ian Barwick)
Йелте Феннема (Jelte Fennema)
Йерун ван дер Хам (Jeroen van der Ham)
Йон Нельсон (Jon Nelson)
КайГай Кохэй (KaiGai Kohei)
Кайл Конрой (Kyle Conroy)
Карен Хаддлстон (Karen Huddleston)
Карл Лехенбауэр (Karl Lehenbauer)
Карл О. Пинц (Karl O. Pinc)
Каспер Жук (Kacper Zuk)
Каталин Якоб (Catalin Iacob)
Кевин Гриттнер (Kevin Grittner)
Ким Росе Карлсен (Kim Rose Carlsen)
Кит Фиске (Keith Fiske)
Клаудио Фрейре (Claudio Freire)
Клинтон Адамс (Clinton Adams)
Конст Чжан (Const Zhang)
Константин Евтеев (Konstantin Evteev)
Константин Книжник (Konstantin Knizhnik)
Константин Пан (Constantin Pan)
Кори Хинкер (Corey Huinker)
Крейг Рингер (Craig Ringer)
Крис Бенди (Chris Bandy)
Крис Ричардс (Chris Richards)
Крис Рупрехт (Chris Ruprecht)
Кристиан Ульрих (Christian Ullrich)
Кристоф Берг (Christoph Berg)
Кунтал Гхош (Kuntal Ghosh)
Курт Карталтепе (Kurt Kartaltepe)
Куэль Чжо (QL Zhuo)
Кётаро Хоригути (Kyotaro Horiguchi)
Лауренц Альбе (Laurenz Albe)

Леонардо Чекки (Leonardo Cecchi)
Лукас Фиттл (Lukas Fittl)
Людовик Вожуа-Пепин (Ludovic Vaugeois-Pepin)
Магнус Хагандер (Magnus Hagander)
Майк Пальмиотто (Mike Palmiotto)
Майкл Дэй (Michael Day)
Максим Милютин (Maksim Milyutin)
Максим Соболев (Maksym Sobolyev)
Марек Чворен (Marek Cvoren)
Марк Дилгер (Mark Dilger)
Марк Кирквуд (Mark Kirkwood)
Марк Петер (Mark Pether)
Марк Расбах (Marc Rassbach)
Марк-Олаф Яшке (Marc-Olaf Jaschke)
Марко Тиикkая (Marko Tiikkaja)
Маркос Кастедо (Marcos Castedo)
Маркус Винанд (Markus Winand)
Марллиус Рибейро (Marllius Ribeiro)
Марти Раудсепп (Marti Raudsepp)
Мартин Маркес (Martín Marqués)
Масахико Савада (Masahiko Sawada)
Матеус Оливейра (Matheus Oliveira)
Мерлин Монкьюр (Merlin Moncure)
Микаэль Пакье (Michael Paquier)
Милош Урбанек (Milos Urbanek)
Митхун Сай (Mithun Cy)
Михаэль Банк (Michael Banck)
Михаэль Мескес (Michael Meskes)
Михаэль Овермайер (Michael Overmeyer)
Мойше Джейкобсон (Moshe Jacobson)
Муртуза Забуавала (Murtuza Zabuawala)
Мэтью Феньяк (Mathieu Fenniak)
Наоки Окано (Naoki Okano)
Натан Боссарт (Nathan Bossart)
Натан Вагнер (Nathan Wagner)
Неха Хатри (Neha Khatri)
Неха Шарма (Neha Sharma)
Никита Глухов (Nikita Glukhov)
Николай Никитин (Nikolay Nikitin)
Николай Шаплов (Nikolay Shaplov)
Николас Бачелли (Nicolas Baccelli)
Николас Гини (Nicolas Guini)
Николас Товен (Nicolas Thauvin)
Николаус Тиль (Nikolaus Thiel)
Никхил Сонтакке (Nikhil Sontakke)
Нил Андерсон (Neil Anderson)
Ной Миш (Noah Misch)
Нориёси Синода (Noriyoshi Shinoda)
Олаф Гавенда (Olaf Gawenda)
Олег Бартунов (Oleg Bartunov)
Оскари Сааренмаа (Oskari Saarenmaa)
Отар Шавадзе (Otar Shavadze)
Паван Деоласи (Pavan Deolasee)
Павел Голубь (Pavel Golub)
Павел Райскуп (Pavel Raiskup)
Павел Стехуле (Pavel Stehule)
Павел Ханак (Pavel Hanák)
Пареш Мор (Paresh More)

Петр Желинек (Petr Jelínek)
Питер Гейган (Peter Geoghegan)
Питер Эйзентраут (Peter Eisentraut)
Пол Рамсей (Paul Ramsey)
Пол Юнгвирт (Paul Jungwirth)
Прабхат Саху (Prabhat Sahu)
Пьер-Эммануэль Андре (Pierre-Emmanuel André)
Пэн Сунь (Peng Sun)
Пётр Стефаняк (Piotr Stefaniak)
Рагнар Оухтерлони (Ragnar Ouchterlony)
Радек Слупик (Radek Slupik)
Раджкумар Рагхуванши (Rajkumar Raghuwanshi)
Райан Мерфи (Ryan Murphy)
Рафа де ла Торре (Rafa de la Torre)
Рафия Сабих (Rafia Sabih)
Рахила Сьед (Rahila Syed)
Регина Обе (Regina Obe)
Ричард Пистоль (Richard Pistole)
Роберт Хаас (Robert Haas)
Робинс Таракан (Robins Tharakan)
Род Тейлор (Rod Taylor)
Роман Шапошник (Roman Shaposhnik)
Рушаб Латиа (Rushabh Lathia)
Саймон Риггс (Simon Riggs)
Сандип Таккар (Sandeep Thakkar)
Свейн Свейнссон (Sveinn Sveinsson)
Свен Р. Кунце (Sven R. Kunze)
Себастьян Луке (Sebastian Luque)
Сергей Бурладян (Sergey Burladyan)
Сергей Копосов (Sergey Koposov)
Сероп Саркуни (Sehrope Sarkuni)
Симоне Готти (Simone Gotti)
Синтия Шан (Cynthia Shang)
Синъити Мацуда (Shinichi Matsuda)
Скотт Милликен (Scott Milliken)
Спенсер Томасон (Spencer Thomason)
Стас Кельвич (Stas Kelvich)
Степан Пестерников (Stepan Pesternikov)
Стив Рэндалл (Steve Randall)
Стив Сингер (Steve Singer)
Стивен Винфилд (Steven Winfield)
Стивен Фрост (Stephen Frost)
Стивен Фэклер (Steven Fackler)
Сурадж Хараге (Suraj Kharage)
Тайки Кондо (Taiki Kondo)
Такаюки Цунакава (Takayuki Tsunakawa)
Такэси Идэриха (Takeshi Ideriha)
Тахир Фахрутдинов (Tahir Fakhroutdinov)
Тацуо Исии (Tatsuo Ishii)
Тацуро Ямада (Tatsuro Yamada)
Тим Гудэйр (Tim Goodaire)
Тобиас Бусман (Tobias Bussmann)
Том Браун (Thom Brown)
Том Дунстан (Tom Dunstan)
Том Лейн (Tom Lane)
Тома ван Тилбург (Tom van Tilburg)
Томас Келлерер (Thomas Kellerer)
Томас Манро (Thomas Munro)

Томаш Вондра (Tomas Vondra)
Томонари Кацумата (Tomonari Katsumata)
Тушар Ахуджа (Tushar Ahuja)
Фабрицио де Ройес Мелло (Fabrízio de Royes Mello)
Фабьен Коэльо (Fabien Coelho)
Фейке Стинберген (Feike Steenbergen)
Феликс Герзаге (Felix Gerzaguet)
Филип Йирсак (Filip Jirsák)
Филипп Бодуэн (Philippe Beaudoin)
Фудзии Масао (Fujii Masao)
Фёдор Сигаев (Teodor Sigaev)
Хайме Казанова (Jaime Casanova)
Ханс Бушман (Hans Buschmann)
Харибабу Комми (Haribabu Kommi)
Хейкки Линнакангас (Heikki Linnakangas)
Хуань Жуань (Huan Ruan)
Чепмен Флэк (Chapman Flack)
Чжоу Дигоал (Zhou Digoal)
Чжэнь Мин Ян (Zhen Ming Yang)
Чуаньтин Ван (Chuanting Wang)
Чхве Ду-Вон (Choi Doo-Won)
Чэнь Хуацзюнь (Chen Huajun)
Шо Като (Sho Kato)
Шон Фаррел (Sean Farrell)
Шэй Роджански (Shay Rojansky)
Эйдзи Сэки (Eiji Seki)
Эйлер Тавейра (Euler Taveira)
Эмил Иггланд (Emil Iggland)
Эмре Хасегели (Emre Hasegeli)
Энди Абелисто (Andy Abelisto)
Эндрю Вилрайт (Andrew Wheelwright)
Эндрю Гирт (Andrew Gierth)
Эндрю Дунстан (Andrew Dunstan)
Энрике Менесес (Enrique Meneses)
Эрвин Брандштеттер (Erwin Brandstetter)
Эрик Нордстром (Erik Nordström)
Эрик Рижкерс (Erik Rijkers)
Эцуро Фудзита (Etsuro Fujita)
Юго Нагата (Yugo Nagata)
Якоб Еггер (Jakob Egger)

E.51. Предыдущие выпуски

Замечания к выпускам предыдущих версий опубликованы на сайте. На момент выпуска Postgres Pro Standard 10 поддерживались следующие версии:

- Postgres Pro Standard 9.6: *https://postgrespro.ru/docs/postgrespro/9.6/release.html*
- Postgres Pro Standard 9.5: *https://postgrespro.ru/docs/postgrespro/9.5/release.html*

Приложение F. Дополнительно поставляемые модули

В этом и следующем приложении содержится информация о дополнительных модулях, которые можно найти в составе дистрибутива Postgres Pro Standard. В их число входят средства портирования, утилиты анализа и подключаемые функции, не включённые в состав основной системы Postgres Pro, в основном потому, что они адресованы ограниченной аудитории или находятся в экспериментальном состоянии, не подходящем для основного дерева кода. Однако это всё не умаляет их полезность.

В этом приложении описываются расширения и другие подключаемые серверные модули. В Приложении G описываются вспомогательные программы.

В Postgres Pro Standard эти модули представлены в виде отдельного пакета `postgrespro-std-10-contrib`.

Многие модули предоставляют дополнительные пользовательские функции, операторы и типы. Чтобы использовать один из таких модулей, когда его исполняемый код установлен, вы должны зарегистрировать новые объекты SQL в СУБД. В Postgres Pro и PostgreSQL версии 9.1 и новее для этого нужно воспользоваться командой CREATE EXTENSION. В чистой базе данных вы можете просто выполнить:

```
CREATE EXTENSION имя_модуля;
```

Запускать эту команду должен суперпользователь баз данных. При этом новые объекты SQL будут зарегистрированы только в текущей базе данных, так что эту команду нужно выполнять в каждой базе данных, в которой вы хотите пользоваться функциональностью этого модуля. Вы также можете запустить её в `template1`, чтобы установленное расширение копировалось во все впоследствии создаваемые базы по умолчанию.

Многие модули позволяют устанавливать свои объекты в схему по выбору. Для этого нужно добавить `SCHEMA` *`имя_схемы`* в команду `CREATE EXTENSION`. По умолчанию объекты устанавливаются в текущую схему для создаваемых объектов, которой по умолчанию становится `public`.

Если ваша база данных была получена в результате выгрузки/перезагрузки данных PostgreSQL версии до 9.1, и вы ранее использовали версию этого модуля, рассчитанную на версию до 9.1, вместо этого вы должны выполнить:

```
CREATE EXTENSION имя_модуля FROM unpackaged;
```

При этом объекты этого модуля версии до 9.1 будут упакованы в соответствующий объект *расширения*. После этого обновления расширения будут осуществляться командой ALTER EXTENSION. За дополнительными сведениями об обновлении расширении обратитесь к Разделу 35.15.

Однако некоторые из этих модулей не являются «расширениями» в этом смысле, а подключаются к серверу по-другому, например, через параметр конфигурации shared_preload_libraries. Подробнее об этом говорится в документации каждого модуля.

F.1. adminpack

Модуль `adminpack` предоставляет несколько вспомогательных функций, которыми могут пользоваться pgAdmin и другие средства администрирования и управления базами данных, например, для удалённого управления файлами журналов сервера. Использовать все эти функции разрешено только суперпользователям.

Функции, приведённые в Таблице F.1, дают доступ на запись к файлам на компьютере, где работает сервер. (См. также функции в Таблице 9.88, которые дают доступ только на чтение.) С их помощью можно обращаться только к файлам в каталоге кластера баз данных, но путь может задаваться и как относительный, и как абсолютный.

Таблица F.1. Функции модуля `adminpack`

Имя	Тип результата	Описание
`pg_catalog.pg_file_write( filename text, data text, append boolean)`	`bigint`	Записать или дописать в текстовый файл
`pg_catalog.pg_file_ rename(oldname text, newname text [, archivename text])`	`boolean`	Переименовать файл
`pg_catalog.pg_file_ unlink(filename text)`	`boolean`	Удалить файл
`pg_catalog.pg_logdir_ls( )`	`setof record`	Получить список файлов журналов в каталоге `log_ directory`

Функция `pg_file_write` записывает данные (*`data`*) в файл с именем *`filename`*. Если флаг *`append`* сброшен, этот файл не должен существовать. Если же флаг *`append`* установлен, существование файла допускается и в этом случае данные будут дописаны в него. Возвращает число записанных байт.

Функция `pg_file_rename` переименовывает файл. Если параметр *`archivename`* опущен или равен NULL, она просто переименовывает файл *`oldname`* в *`newname`* (файл с новым именем не должен существовать). Если параметр *`archivename`* задан, она сначала переименовывает *`newname`* в *`archivename`* (такой файл не должен существовать), а затем переименовывает *`oldname`* в *`newname`*. В случае ошибки на втором этапе переименования она попытается переименовать *`archivename`* назад в *`newname`*, прежде чем выдать ошибку. Возвращает true в случае успеха и false, если исходные файлы отсутствуют или их невозможно изменить; в других случаях выдаются ошибки.

Функция `pg_file_unlink` удаляет заданный файл. Возвращает true в случае успеха, false в случае отсутствия указанного файла либо при сбое в вызове `unlink()`; в других случаях выдаются ошибки.

Функция `pg_logdir_ls` возвращает время создания и пути всех файлов журналов в каталоге log_directory. Чтобы эта функция работала, параметр log_filename должен иметь значение по умолчанию (`postgresql-%Y-%m-%d_%H%M%S.log`).

Функции, перечисленные в Таблице F.2, являются устаревшими и не должны использоваться в новых приложениях; вместо них следует использовать функции, описанные в Таблице 9.78 и Таблице 9.88. Эти функции включены в `adminpack` только для совместимости со старыми версиями pgAdmin.

Таблица F.2. Устаревшие функции модуля `adminpack`

Имя	Тип результата	Описание
`pg_catalog.pg_file_read( filename text, offset bigint, nbytes bigint)`	`text`	Альтернативный вариант функции `pg_read_file()`
`pg_catalog.pg_file_ length(filename text)`	`bigint`	Выдаёт то же значение, что содержится в столбце `size` результата `pg_stat_file()`
`pg_catalog.pg_logfile_ rotate()`	`integer`	Альтернативный вариант `pg_ rotate_logfile()`, возвращает целое (0 или 1), а не значение `boolean`

F.2. amcheck

Модуль amcheck предоставляет функции, позволяющие проверять логическую целостность структуры отношений. Если нарушения структуры не обнаруживаются, эти функции отрабатывают без ошибок.

Эти функции проверяют различные *инварианты* в структуре представления определённых отношений. Правильность работы функций методов доступа, стоящих за сканированием индекса и другими важными операциями, зависит от всегда соблюдаемых инвариантов. Например, определённые функции проверяют, помимо остальных вещей, что все страницы B-дерева содержат элементы в «логическом» порядке (например, индекс-B-дерево, построенный по столбцу text, должен содержать кортежи, упорядоченные в лексическом порядке с учётом правила сортировки). Если этот конкретный инвариант каким-то образом нарушается, следует ожидать, что бинарный поиск на затронутой странице введёт в заблуждение процедуру сканирования индекса, что приведёт к неверным результатам запросов SQL.

Проверка выполняется теми же процедурами, что используются при сканировании индекса, и это может быть код пользовательского класса операторов. Например, проверка индекса-B-дерева задействует сравнения, выполняемые одной или несколькими опорными функциями B-дерева под номером 1. Подробнее опорные функции класса операторов описываются в Подразделе 35.14.3.

Функции amcheck могут выполнять только суперпользователи.

F.2.1. Функции

bt_index_check(index regclass, heapallindexed boolean) returns void

bt_index_check проверяет, соблюдаются ли в целевом индексе-B-дереве различные инварианты. Пример использования:

```
test=# SELECT bt_index_check(index => c.oid, heapallindexed => i.indisunique),
               c.relname,
               c.relpages
FROM pg_index i
JOIN pg_opclass op ON i.indclass[0] = op.oid
JOIN pg_am am ON op.opcmethod = am.oid
JOIN pg_class c ON i.indexrelid = c.oid
JOIN pg_namespace n ON c.relnamespace = n.oid
WHERE am.amname = 'btree' AND n.nspname = 'pg_catalog'
-- Не проверять временные таблицы (они могут относиться к другим сеансам):
AND c.relpersistence != 't'
-- Функция может выдать ошибку без этих условий:
AND c.relkind = 'i' AND i.indisready AND i.indisvalid
ORDER BY c.relpages DESC LIMIT 10;
 bt_index_check |             relname             | relpages
----------------+---------------------------------+----------
                | pg_depend_reference_index       |       43
                | pg_depend_depender_index        |       40
                | pg_proc_proname_args_nsp_index  |       31
                | pg_description_o_c_o_index      |       21
                | pg_attribute_relid_attnam_index |       14
                | pg_proc_oid_index               |       10
                | pg_attribute_relid_attnum_index |        9
                | pg_amproc_fam_proc_index        |        5
                | pg_amop_opr_fam_index           |        5
                | pg_amop_fam_strat_index         |        5
(10 rows)
```

Этот пример демонстрирует сеанс проверки 10 самых больших индексов системных каталогов в базе данных «test». Проверка всех кортежей кучи на предмет наличия соответствующих кортежей индекса запрашивается только для тех из этих индексов, которые являются

уникальными. Так как ошибки не было, все проверенные индексы представляются логически целостными. Естественно, этот запрос можно легко изменить, чтобы функция `bt_index_check` вызывалась для всех индексов в базе, которые поддерживают эту проверку.

Функция `bt_index_check` запрашивает блокировку `AccessShareLock` для целевого индекса и отношения, которому он принадлежит. Это тот же режим блокировки, что запрашивается для отношений обычными операторами `SELECT`. `bt_index_check` не проверяет инварианты, существующие в иерархии потомок/родитель, но проверяет представление всех кортежей кучи в индексе в виде индексных кортежей, когда параметр *`heapallindexed`* равен `true`. Когда в работающей производственной среде требуется регулярная лёгкая проверка на наличие нарушений, использование `bt_index_check` часто будет подходящим компромиссом между полнотой проверки и минимизацией влияния на производительность и доступность приложения.

`bt_index_parent_check(index regclass, heapallindexed boolean) returns void`

Функция `bt_index_parent_check` проверяет, соблюдаются ли в целевом объекте, индексе-B-дереве, различные инварианты. Кроме того, если аргумент *`heapallindexed`* равен `true`, эта функция проверяет наличие в индексе всех кортежей из кучи, которые должны в него попасть, и отсутствие потерянных связей в структуре индекса. Проверки, которые может производить `bt_index_parent_check`, включают в себя все проверки, выполняемые функцией `bt_index_check`. Функцию `bt_index_parent_check` можно считать более полноценным вариантом `bt_index_check`: в отличие от `bt_index_check`, `bt_index_parent_check` проверяет ещё и инварианты, существующие в иерархии родитель/потомок. `bt_index_parent_check` следует общему соглашению и выдаёт ошибку в случае обнаружения логической несогласованности или другой проблемы.

Функция `bt_index_parent_check` запрашивает в целевом индексе блокировку `ShareLock` (также `ShareLock` запрашивается и в основном отношении). Эти блокировки предотвращают одновременное изменение данных командами `INSERT`, `UPDATE` и `DELETE`. Эти блокировки также препятствуют одновременной обработке нижележащего отношения командой `VACUUM` и другими вспомогательными командами. Заметьте, что эта функция удерживает блокировки только во время выполнения, а не на протяжении всей транзакции.

Дополнительные проверки, проводимые функцией `bt_index_parent_check`, более ориентированы на выявление различных патологических случаев. В том числе это может быть неправильно реализованный класс операторов B-дерева, используемый проверяемым индексом, или, гипотетически, неизвестные ошибки в нижележащем коде метода доступа индекса-B-дерева. Заметьте, что функцию `bt_index_parent_check` нельзя применять, когда включён режим горячего резерва (то есть на физических репликах в режиме «только чтение»), в отличие от `bt_index_check`.

F.2.2. Дополнительная проверка *`heapallindexed`*

Когда аргумент *`heapallindexed`* проверяющих функций равен `true`, для таблицы, связанной с отношением целевого индекса, добавляется дополнительная фаза проверки. Она включает «фиктивную» операцию `CREATE INDEX`, которая проверяет присутствие всех гипотетических новых индексных кортежей по временной сводной структуре в памяти (она создаётся при необходимости на первом этапе проверки). Сводная структура «помечает» каждый кортеж, который находится в целевом индексе. На высоком уровне идея проверки *`heapallindexed`* состоит в том, чтобы убедиться, что новый индекс, равнозначный целевому, содержит только те записи, которые можно найти в существующей структуре.

С дополнительным этапом *`heapallindexed`* связаны значительные издержки: проверка обычно будет выполняться в несколько раз дольше. Однако никакие новые блокировки уровня отношения при проверке *`heapallindexed`* не запрашиваются.

Сводная структура ограничивается по объёму значением `maintenance_work_mem`. Для выявления несогласованности в представленных в индексе кортежах с вероятностью упущений в пределах

2% требуется приблизительно 2 байта памяти на кортеж. По мере уменьшения объёма памяти в пересчёте на кортеж этот процент медленно растёт. Этот подход значительно ограничивает издержки такой проверки, и при этом лишь немного уменьшается вероятность выявления проблемы, особенно в инсталляциях, где эта проверка включена в процедуру регулярного обслуживания. Даже если единичное отсутствие или повреждение кортежа упущено, есть все шансы выявить его при очередной проверке.

F.2.3. Эффективное использование `amcheck`

Модуль `amcheck` может быть полезен для выявления различных типов проблем, которые могут остаться незамеченными при включении контрольных сумм страниц данных. В частности это:

- Структурные несоответствия, возникающие при некорректной реализации класса операторов.

  В том числе это проблемы, возникающие при изменении правил сравнения в операционной системе. Сравнения данных сортируемого типа, например `text`, должны быть постоянными (как и все сравнения, применяемые при сканировании индекса-B-дерева), что подразумевает неизменность правил сортировки в операционной системе. Проблемы могут возникать при обновлениях правил в операционной системе, хотя такие случаи редки. Чаще проявляются несоответствия порядка сортировки между ведущим и ведомым сервером, например, из-за различий *основных* версий используемых операционных систем. Возникающие расхождения обычно наблюдаются только на ведомых серверах, так что и выявить их обычно можно только на них.

  Когда возникает подобная проблема, она может затрагивать не абсолютно все индексы, построенные с порочным правилом сортировки, просто потому что *индексированные* значения могут иметь тот же абсолютный порядок, независящий от различий поведения. За дополнительными сведениями об использовании в Postgres Pro правил сортировки и локалей операционной системы обратитесь к Разделу 22.1 и Разделу 22.2.

- Несоответствия структуры между индексами и проиндексированными отношениями в куче (когда выполняется проверка *`heapallindexed`*).

  Во время обычных операций перекрёстная проверка индексов по отношениям в куче не производится. Симптомы повреждения данных в куче могут быть неочевидными.

- Повреждения, вызванные гипотетическими неизвестными ошибками в нижележащем коде методов доступа, коде сортировки и управления транзакциями Postgres Pro.

  Автоматическая проверка структурной целостности индексов играет важную роль в общем тестировании новых или предлагаемых возможностей Postgres Pro, с которыми может возникнуть логическая несогласованность. Такую же роль играет проверка структуры таблицы и связанной информации о видимости и состоянии транзакций. И поэтому одна из очевидных стратегий тестирования — регулярно вызывать функции `amcheck` при проведении стандартных регрессионных тестов.

- Ошибки в файловой системе или подсистеме хранения, когда просто не включены контрольные суммы.

  Заметьте, что `amcheck` рассматривает страницу в том виде, как она представлена в некотором буфере разделяемой памяти к моменту проверки, если при обращению к нужному блоку он уже находится в разделяемом буфере. Вследствие этого, `amcheck` не обязательно видит данные, находящиеся в файловой системе в момент проверки. Заметьте, что когда контрольные суммы включены, `amcheck` может выдать ошибку из-за несоответствия контрольных сумм, если в буфер будет считываться испорченный блок.

- Повреждения, вызванные дефектными чипами ОЗУ или вообще подсистемой памяти.

  Postgres Pro не защищает от ошибок памяти; предполагается, что в эксплуатируемом вами сервере установлена память с ECC (Error Correcting Codes, Коды исправления ошибок) или лучшая защита. Однако память ECC обычно защищает только от ошибок в одном бите и не следует считать её *абсолютной* защитой от сбоев, приводящих к повреждению памяти.

Когда выполняется проверка *heapallindexed*, в целом значительно увеличивается шанс выявления ошибок в отдельных битах, так как она тестирует точное двоичное равенство и сверяет проиндексированные атрибуты с кучей.

Вообще говоря, amcheck может доказать только наличие повреждений, но не доказать их отсутствие.

F.2.4. Исправление повреждений

Когда amcheck сигнализирует о повреждении данных, ложные срабатывания практически исключены. amcheck считает ошибочными ситуации, которые никогда не должны наблюдаться по определению, поэтому ошибки amcheck, как правило, требуют тщательного анализа.

Общего метода устранения проблем, которые может выявить amcheck, не существует. Начать нужно с поиска корня проблемы, приводящей к нарушению инварианта. Полезную роль в диагностике повреждений, которые выявляет amcheck, может сыграть pageinspect. Одна лишь команда REINDEX может быть неэффективна, когда потребуется исправить повреждения.

F.3. auth_delay

Модуль auth_delay добавляет небольшую задержку в процессе проверки подлинности перед тем, как выдаётся сообщение об ошибке, чтобы усложнить подбор паролей к базам данных. Заметьте, что это никоим образом не препятствует атакам типа «отказ в обслуживании», а даже наоборот, может помочь их осуществить, так как процессы, ожидающие сообщения об ошибке, всё равно занимают слоты подключения.

Чтобы эта функция работала, данный модуль нужно загрузить посредством параметра конфигурации shared_preload_libraries в postgresql.conf.

F.3.1. Параметры конфигурации

auth_delay.milliseconds (int)

Число миллисекунд, которое нужно подождать, прежде чем сообщать об ошибке аутентификации. По умолчанию 0.

Эти параметры должны задаваться в postgresql.conf. Обычное использование выглядит так:

```
# postgresql.conf
shared_preload_libraries = 'auth_delay'

auth_delay.milliseconds = '500'
```

F.3.2. Автор

КайГай Кохэй <kaigai@ak.jp.nec.com>

F.4. auto_explain

Модуль auto_explain предоставляет возможность автоматического протоколирования планов выполнения медленных операторов, что позволяет обойтись без выполнения EXPLAIN вручную. Это особенно полезно для выявления неоптимизированных запросов в больших приложениях.

Этот модуль не предоставляет функций, доступных из SQL. Чтобы использовать его, просто загрузите его в процесс сервера. Это можно сделать в отдельном сеансе:

```
LOAD 'auto_explain';
```

(Для этого нужно быть суперпользователем.) Более типична конфигурация, когда он загружается в некоторые или все сеансы в результате включения auto_explain в переменную session_preload_libraries или в shared_preload_libraries в файле postgresql.conf. Загрузив этот

модуль, вы можете отслеживать исключительно медленные запросы, вне зависимости от того, когда они происходят. Конечно, это имеет свою цену.

F.4.1. Параметры конфигурации

Есть несколько параметров конфигурации, которые управляют поведением `auto_explain`. Заметьте, что поведение по умолчанию сводится к бездействию, так что необходимо установить как минимум переменную `auto_explain.log_min_duration`, если вы хотите получить какие-либо результаты.

`auto_explain.log_min_duration` (`integer`)

Переменная `auto_explain.log_min_duration` задаёт время выполнения оператора, в миллисекундах, при превышении которого план оператора будет протоколироваться. Если это значение равно 0, протоколироваться будут планы всех операторов. При значении -1 (по умолчанию) протоколирование планов полностью отключается. Например, если вы установите значение `250ms`, протоколироваться будут все запросы, выполняющиеся 250 мс и дольше. Изменить этот параметр могут только суперпользователи.

`auto_explain.log_analyze` (`boolean`)

При включении параметра `auto_explain.log_analyze` в протокол будет записываться вывод команды `EXPLAIN ANALYZE`, а не простой `EXPLAIN`. По умолчанию этот параметр отключён. Изменить его могут только суперпользователи.

> **Примечание**
>
> Когда этот параметр включён, замер времени на уровне узлов плана производится для всех операторов, даже если они выполняются недостаточно долго для протоколирования. Это может оказать крайне негативное влияние на производительность. Отключение `auto_explain.log_timing` исключает это влияние, но при этом собирается меньше информации.

`auto_explain.log_buffers` (`boolean`)

Параметр `auto_explain.log_buffers` определяет, будет ли при протоколировании плана выполнения выводиться статистика об использовании буферов; он равносилен указанию `BUFFERS` команды `EXPLAIN`. Этот параметр действует, только если включён параметр `auto_explain.log_analyze`. По умолчанию этот параметр отключён. Изменить его могут только суперпользователи.

`auto_explain.log_timing` (`boolean`)

Параметр `auto_explain.log_timing` определяет, будет ли при протоколировании плана выполнения выводиться длительность на уровне узлов: он равнозначен указанию `TIMING` команды `EXPLAIN`. Издержки от постоянного чтения системных часов могут значительно замедлить запросы в некоторых системах, так что может иметь смысл отключать этот параметр, когда нужно знать только знать количество строк, но не точную длительность каждого узла. Этот параметр действует, только если включён `auto_explain.log_analyze`. По умолчанию этот параметр отключён. Изменить его могут только суперпользователи.

`auto_explain.log_triggers` (`boolean`)

При включении параметра `auto_explain.log_triggers` в протокол будет записываться статистика выполнения триггеров. Этот параметр действует, только если включён параметр `auto_explain.log_analyze`. По умолчанию этот параметр отключён. Изменить его могут только суперпользователи.

`auto_explain.log_verbose` (`boolean`)

Параметр `auto_explain.log_verbose` определяет, будут ли при протоколировании плана выполнения выводиться подробные сведения; он равнозначен указанию `VERBOSE` команды

EXPLAIN. По умолчанию этот параметр отключён. Изменить его могут только суперпользователи.

auto_explain.log_format (enum)

Параметр auto_explain.log_format выбирает формат вывода для EXPLAIN. Он может принимать значение text, xml, json и yaml. Значение по умолчанию — text. Изменить этот параметр могут только суперпользователи.

auto_explain.log_nested_statements (boolean)

При включении параметра auto_explain.log_nested_statements протоколированию могут подлежать и вложенные операторы (операторы, выполняемые внутри функции). Когда он отключён, протоколируются планы запросов только верхнего уровня. Изменить этот параметр могут только суперпользователи.

auto_explain.sample_rate (real)

Параметр auto_explain.sample_rate задаёт для auto_explain процент операторов, которые будут отслеживаться в каждом сеансе. Значение по умолчанию — 1, то есть отслеживаются все запросы. Вложенные операторы отслеживаются совместно — либо все, либо никакой из них. Изменить этот параметр могут только суперпользователи.

В обычной ситуации эти параметры устанавливаются в postgresql.conf, хотя суперпользователи могут изменить их «на лету» в рамках своих сеансов. Типичное их использование может выглядеть так:

```
# postgresql.conf
session_preload_libraries = 'auto_explain'

auto_explain.log_min_duration = '3s'
```

F.4.2. Пример

```
postgres=# LOAD 'auto_explain';
postgres=# SET auto_explain.log_min_duration = 0;
postgres=# SET auto_explain.log_analyze = true;
postgres=# SELECT count(*)
           FROM pg_class, pg_index
           WHERE oid = indrelid AND indisunique;
```

В результате этих команд может быть получен такой вывод:

```
LOG:  duration: 0.196 ms planning: 0.548 ms plan:
  Query Text: SELECT count(*)
              FROM pg_class, pg_index
              WHERE oid = indrelid AND indisunique;
  Aggregate  (cost=16.79..16.80 rows=1 width=0) (actual time=3.626..3.627 rows=1
 loops=1)
    ->  Hash Join  (cost=4.17..16.55 rows=92 width=0) (actual time=3.349..3.594 rows=92
 loops=1)
          Hash Cond: (pg_class.oid = pg_index.indrelid)
          ->  Seq Scan on pg_class  (cost=0.00..9.55 rows=255 width=4) (actual
 time=0.016..0.140 rows=255 loops=1)
          ->  Hash  (cost=3.02..3.02 rows=92 width=4) (actual time=3.238..3.238 rows=92
 loops=1)
                Buckets: 1024  Batches: 1  Memory Usage: 4kB
                ->  Seq Scan on pg_index  (cost=0.00..3.02 rows=92 width=4) (actual
 time=0.008..3.187 rows=92 loops=1)
                      Filter: indisunique
```

Заметьте, что в Postgres Pro было добавлено новое поле для отображения длительности планирования.

F.4.3. Авторы

Такахиро Итагаки <itagaki.takahiro@oss.ntt.co.jp>

Postgres Professional, Москва, Россия

F.5. bloom

Модуль bloom предоставляет индексный метод доступа, основанный на *фильтрах Блума*.

Фильтр Блума представляет собой компактную структуру данных, позволяющую проверить, является ли элемент членом множества. В виде метода доступа индекса он позволяет быстро исключать неподходящие кортежи по сигнатурам, размер которых определяется при создании индекса.

Сигнатура — это неточное представление проиндексированных атрибутов, вследствие чего оно допускает ложные положительные срабатывания; то есть оно может показывать, что элемент содержится в множестве, хотя это не так. Поэтому результаты поиска по такому индексу должны всегда перепроверяться по фактическим значениям атрибутов записи в таблице. Чем больше размер сигнатуры, тем меньше вероятность ложного срабатывания и число напрасных обращений к таблице, но это, разумеется, влечёт увеличение индекса и замедление сканирования.

Этот тип индекса наиболее полезен, когда в таблице много атрибутов и в запросах проверяются их произвольные сочетания. Традиционный индекс-B-дерево быстрее индекса Блума, но для поддержки всевозможных запросов может потребоваться множество индексов типа B-дерево, при том что индекс Блума нужен всего один. Заметьте, однако, что индексы Блума поддерживают только проверки на равенство, тогда как индексы-B-деревья также полезны при проверке неравенств и поиске в диапазоне.

F.5.1. Параметры

Индекс bloom принимает в своём предложении WITH следующие параметры:

length

Длина каждой сигнатуры (элемента индекса) в битах, округлённая вверх до ближайшего числа, кратного 16. Значение по умолчанию — 80, а максимальное значение — 4096.

col1 — col32

Число битов, генерируемых для каждого столбца индекса. В имени параметра отражается номер столбца индекса, для которого это число задаётся. Значение по умолчанию — 2 бита, а максимум — 4095. Параметры для неиспользуемых столбцов индекса игнорируются.

F.5.2. Примеры

Пример создания индекса bloom:

```
CREATE INDEX bloomidx ON tbloom USING bloom (i1,i2,i3)
       WITH (length=80, col1=2, col2=2, col3=4);
```

Эта команда создаёт индекс с длиной сигнатуры 80 бит, в которой атрибуты i1 и i2 отображаются в 2 бита, а атрибут i3 — в 4. Мы могли бы опустить указания length, col1 и col2, так как в них задаются значения по умолчанию.

Ниже представлен более полный пример определения и использования индекса Блума, а также приводится сравнение его с равнозначным индексом-B-деревом. Видно, что индекс Блума значительно меньше индекса-B-дерева, и при этом он может работать быстрее.

```
=# CREATE TABLE tbloom AS
   SELECT
     (random() * 1000000)::int as i1,
```

```
     (random() * 1000000)::int as i2,
     (random() * 1000000)::int as i3,
     (random() * 1000000)::int as i4,
     (random() * 1000000)::int as i5,
     (random() * 1000000)::int as i6
   FROM
  generate_series(1,10000000);
SELECT 10000000
```

Последовательное сканирование по этой большой таблице выполняется долго:

```
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                                    QUERY PLAN

-------------------------------------------------------------------------------------------------
 Seq Scan on tbloom  (cost=0.00..2137.14 rows=3 width=24) (actual time=14.894..14.895
 rows=0 loops=1)
   Filter: ((i2 = 898732) AND (i5 = 123451))
   Rows Removed by Filter: 100000
 Planning Time: 0.350 ms
 Execution Time: 14.916 ms
(5 rows)
```

Даже при наличии индекса btree сканирование остаётся последовательным:

```
=# CREATE INDEX btreeidx ON tbloom (i1, i2, i3, i4, i5, i6);
CREATE INDEX
=# SELECT pg_size_pretty(pg_relation_size('btreeidx'));
 pg_size_pretty
----------------
 3992 kB
(1 row)
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                                    QUERY PLAN

-------------------------------------------------------------------------------------------------
 Seq Scan on tbloom  (cost=0.00..2137.00 rows=2 width=24) (actual time=12.004..12.004
 rows=0 loops=1)
   Filter: ((i2 = 898732) AND (i5 = 123451))
   Rows Removed by Filter: 100000
 Planning Time: 0.157 ms
 Execution Time: 12.019 ms
(5 rows)
```

Если же для таблицы создан индекс bloom, поиск такого рода выполняется эффективнее, чем с индексом btree:

```
=# CREATE INDEX bloomidx ON tbloom USING bloom (i1, i2, i3, i4, i5, i6);
CREATE INDEX
=# SELECT pg_size_pretty(pg_relation_size('bloomidx'));
 pg_size_pretty
----------------
 1584 kB
(1 row)
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                                          QUERY PLAN

-------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on tbloom  (cost=1792.00..1799.69 rows=2 width=24) (actual
 time=0.390..0.391 rows=0 loops=1)
```

```
   Recheck Cond: ((i2 = 898732) AND (i5 = 123451))
   Rows Removed by Index Recheck: 19
   Heap Blocks: exact=19
   ->  Bitmap Index Scan on bloomidx  (cost=0.00..1792.00 rows=2 width=0) (actual
 time=0.361..0.361 rows=19 loops=1)
         Index Cond: ((i2 = 898732) AND (i5 = 123451))
 Planning Time: 0.128 ms
 Execution Time: 0.415 ms
(8 rows)
```

При таком подходе основная проблема поиска по B-дереву состоит в том, что B-дерево неэффективно, когда условия поиска не ограничивают ведущие столбцы индекса. Поэтому, применяя индексы типа B-дерево, лучше создавать отдельные индексы для каждого столбца. В этом случае планировщик построит примерно такой план:

```
=# CREATE INDEX btreeidx1 ON tbloom (i1);
CREATE INDEX
=# CREATE INDEX btreeidx2 ON tbloom (i2);
CREATE INDEX
=# CREATE INDEX btreeidx3 ON tbloom (i3);
CREATE INDEX
=# CREATE INDEX btreeidx4 ON tbloom (i4);
CREATE INDEX
=# CREATE INDEX btreeidx5 ON tbloom (i5);
CREATE INDEX
=# CREATE INDEX btreeidx6 ON tbloom (i6);
CREATE INDEX
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                                          QUERY PLAN

------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on tbloom  (cost=24.34..32.03 rows=2 width=24) (actual
 time=0.031..0.032 rows=0 loops=1)
   Recheck Cond: ((i5 = 123451) AND (i2 = 898732))
   ->  BitmapAnd  (cost=24.34..24.34 rows=2 width=0) (actual time=0.028..0.029 rows=0
 loops=1)
         ->  Bitmap Index Scan on btreeidx5  (cost=0.00..12.04 rows=500 width=0)
 (actual time=0.028..0.028 rows=0 loops=1)
               Index Cond: (i5 = 123451)
         ->  Bitmap Index Scan on btreeidx2  (cost=0.00..12.04 rows=500 width=0) (never
 executed)
               Index Cond: (i2 = 898732)
 Planning Time: 0.474 ms
 Execution Time: 0.064 ms
(9 rows)
```

Хотя этот запрос выполняется гораздо быстрее, чем с каким-либо одиночным индексом, мы платим за это увеличением размера индекса. Каждый индекс-B-дерево занимает 2 Мбайта, так что общий объём индексов составляет 12 Мбайт, что в 8 раз больше размера индекса Блума.

F.5.3. Интерфейс класса операторов

Класс операторов для индексов Блума требует наличия только хеш-функции для индексируемого типа данных и оператора равенства для поиска. Этот пример демонстрирует соответствующее определение класса операторов для типа `text`:

```
CREATE OPERATOR CLASS text_ops
DEFAULT FOR TYPE text USING bloom AS
    OPERATOR    1   =(text, text),
    FUNCTION    1   hashtext(text);
```

F.5.4. Ограничения

- В этот модуль включены только классы операторов для `int4` и `text`.
- При поиске поддерживается только оператор `=`. Но в будущем возможно добавление поддержки для массивов с операциями объединения и пересечения.
- Метод доступа `bloom` не поддерживает уникальные индексы (`UNIQUE`).
- Метод доступа `bloom` не поддерживает поиск значений `NULL`.

F.5.5. Авторы

Фёдор Сигаев `<teodor@postgrespro.ru>`, Postgres Professional, Москва, Россия

Александр Коротков `<a.korotkov@postgrespro.ru>`, Postgres Professional, Москва, Россия

Олег Бартунов `<obartunov@postgrespro.ru>`, Postgres Professional, Москва, Россия

F.6. btree_gin

Модуль `btree_gin` предоставляет показательные классы операторов GIN, реализующие поведение, подобное тому, что реализуют обычные классы B-дерева, для типов данных `int2`, `int4`, `int8`, `float4`, `float8`, `timestamp with time zone`, `timestamp without time zone`, `time with time zone`, `time without time zone`, `date`, `interval`, `oid`, `money`, `"char"`, `varchar`, `text`, `bytea`, `bit`, `varbit`, `macaddr`, `macaddr8`, `inet`, `cidr` и всех типов `enum`.

Вообще говоря, эти классы операторов не будут работать быстрее аналогичных стандартных методов индекса-B-дерева, и им не хватает одной важной возможности стандартной реализации B-дерева: возможности ограничивать уникальность. Тем не менее их можно применять для тестирования GIN или взять за основу для разработки других классов операторов GIN. Также, для запросов, где проверяется и столбец с индексом GIN, и столбец с индексом-B-деревом, может быть более эффективным создать составной индекс GIN, который использует один из этих классов операторов, чем использовать два отдельных индекса, выборку из которых придётся объединять, вычисляя AND битовых карт.

F.6.1. Пример использования

```
CREATE TABLE test (a int4);
-- создать индекс
CREATE INDEX testidx ON test USING GIN (a);
-- запрос
SELECT * FROM test WHERE a < 10;
```

F.6.2. Авторы

Фёдор Сигаев (`<teodor@stack.net>`) и Олег Бартунов (`<oleg@sai.msu.su>`). Подробности можно найти на странице *http://www.sai.msu.su/~megera/oddmuse/index.cgi/Gin*.

F.7. btree_gist

Модуль `btree_gist` предоставляет показательные классы операторов GiST, реализующие поведение, подобное тому, что реализуют обычные классы B-дерева, для типов данных `int2`, `int4`, `int8`, `float4`, `float8`, `numeric`, `timestamp with time zone`, `timestamp without time zone`, `time with time zone`, `time without time zone`, `date`, `interval`, `oid`, `money`, `char`, `varchar`, `text`, `bytea`, `bit`, `varbit`, `macaddr`, `macaddr8`, `inet`, `cidr`, `uuid` и всех типов `enum`.

Вообще говоря, эти классы операторов не будут работать быстрее аналогичных стандартных методов индекса-B-дерева, и им не хватает одной важной возможности стандартной реализации B-дерева: возможности ограничивать уникальность. Однако они предлагают несколько других возможностей, описанных ниже. Также эти классы операторов полезны, когда требуется составной индекс GiST, в котором некоторые столбцы имеют типы данных, индексируемые только

с GiST, а другие — простые типы. Наконец, эти классы операторов можно применять для тестирования GiST или взять за основу для разработки других классов операторов GiST.

Помимо типичных операторов поиска по B-дереву, `btree_gist` также поддерживает использование индекса для операции `<>` («не равно»). Это может быть полезно в сочетании с ограничением-исключением, как описано ниже.

Также, для типов данных, имеющих естественную метрику расстояния, `btree_gist` определяет оператор расстояния `<->` и поддерживает использование индексов GiST для поиска ближайших соседей с применением этого оператора. Операторы расстояния определены для типов `int2`, `int4`, `int8`, `float4`, `float8`, `timestamp with time zone`, `timestamp without time zone`, `time without time zone`, `date`, `interval`, `oid` и `money`.

F.7.1. Пример использования

Простой пример использования `btree_gist` вместо `btree`:

```
CREATE TABLE test (a int4);
-- создать индекс
CREATE INDEX testidx ON test USING GIST (a);
-- запрос
SELECT * FROM test WHERE a < 10;
-- поиск ближайших соседей: найти десять записей, ближайших к "42"
SELECT *, a <-> 42 AS dist FROM test ORDER BY a <-> 42 LIMIT 10;
```

Так можно использовать ограничение-исключение, состоящее в том, что в клетке в зоопарке могут содержаться животные только одного типа:

```
=> CREATE TABLE zoo (
  cage   INTEGER,
  animal TEXT,
  EXCLUDE USING GIST (cage WITH =, animal WITH <>)
);

=> INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=> INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=> INSERT INTO zoo VALUES(123, 'lion');
ERROR:  conflicting key value violates exclusion constraint "zoo_cage_animal_excl"
DETAIL:  Key (cage, animal)=(123, lion) conflicts with existing key (cage,
 animal)=(123, zebra).
=> INSERT INTO zoo VALUES(124, 'lion');
INSERT 0 1
```

F.7.2. Авторы

Фёдор Сигаев (<teodor@stack.net>), Олег Бартунов (<oleg@sai.msu.su>), Янко Рихтер (<jankorichter@yahoo.de>) и Пол Юнгвирт (<pj@illuminatedcomputing.com>). Подробности можно найти на странице *http://www.sai.msu.su/~megera/postgres/gist/*.

F.8. chkpass

Этот модуль реализует тип данных `chkpass`, предназначенный для хранения зашифрованных паролей. Каждый пароль автоматически преобразуется в зашифрованный вид при вводе и всегда хранится зашифрованным. Для проверки пароля его нужно сравнить с паролем в открытом виде, который будет также зашифрован перед сравнением.

В коде предусмотрена возможность выдавать ошибку, если вводимый пароль оказывается слишком простым. Однако в настоящее время это просто заглушка, которая ничего не делает.

Если вводимая строка начинается с двоеточия, предполагается, что это уже зашифрованный пароль и он сохраняется без дополнительного шифрования. Это позволяет сохранять ранее зашифрованные пароли.

Выводимая строка этого типа предваряется двоеточием. Это позволяет выгружать и заново загружать пароли, не расшифровывая их. Если вы хотите получить строку зашифрованного пароля без двоеточия, можно использовать функцию `raw()`. Благодаря этому, данный тип можно применять, например, с модулем `Auth_PostgreSQL` для Apache.

Для шифрования используется стандартная функция Unix `crypt()`, так что на данную реализацию распространяются все обычные ограничения этой функции; в частности, учитываются только первые восемь символов пароля.

Заметьте, что тип данных `chkpass` не является индексируемым.

Пример использования:

```
test=# create table test (p chkpass);
CREATE TABLE
test=# insert into test values ('hello');
INSERT 0 1
test=# select * from test;
       p
----------------
 :dVGkpXdOrE3ko
(1 row)

test=# select raw(p) from test;
      raw
---------------
 dVGkpXdOrE3ko
(1 row)

test=# select p = 'hello' from test;
 ?column?
----------
 t
(1 row)

test=# select p = 'goodbye' from test;
 ?column?
----------
 f
(1 row)
```

F.8.1. Автор

Д'Арси Дж. М. Каин (<darcy@druid.net>)

F.9. citext

Модуль `citext` предоставляет тип данных для строк, нечувствительных к регистру, `citext`. По сути он сравнивает значения, вызывая внутри себя функцию `lower`. В остальном он почти не отличается от типа `text`.

F.9.1. Обоснование

Стандартный способ выполнить сравнение строк без учёта регистра в Postgres Pro заключается в использовании функции `lower` при сравнении значений, например

```
SELECT * FROM tab WHERE lower(col) = LOWER(?);
```

Этот подход работает довольно хорошо, но имеет ряд недостатков:

- Операторы SQL становятся громоздкими, и нужно не забывать всегда обрабатывать функцией `lower` и столбец, и значение.
- Индекс не будет использоваться, если только дополнительно не создать функциональный индекс с функцией `lower`.
- Если вы объявляете столбец как `UNIQUE` или `PRIMARY KEY`, неявно создаваемый индекс будет чувствительным к регистру. Поэтому он бесполезен для регистронезависимого поиска, так же как он не будет обеспечивать уникальность без учёта регистра.

Тип данных `citext` позволяет исключить вызовы `lower` в SQL-запросах и позволяет сделать первичный ключ регистронезависимым. Тип `citext` учитывает локаль, так же, как и тип `text`, что означает, что сравнение символов в верхнем и нижнем регистре зависит от правил `LC_CTYPE` для базы данных. Это поведение, опять же, не отличается от вызовов `lower` в запросах. Но так как оно реализуется прозрачно типом данных, в самих запросах дополнительно не нужно ничего делать.

F.9.2. Как его использовать

Простой пример использования:

```
CREATE TABLE users (
    nick CITEXT PRIMARY KEY,
    pass TEXT   NOT NULL
);

INSERT INTO users VALUES ( 'larry',  md5(random()::text) );
INSERT INTO users VALUES ( 'Tom',    md5(random()::text) );
INSERT INTO users VALUES ( 'Damian', md5(random()::text) );
INSERT INTO users VALUES ( 'NEAL',   md5(random()::text) );
INSERT INTO users VALUES ( 'Bjørn',  md5(random()::text) );

SELECT * FROM users WHERE nick = 'Larry';
```

Оператор `SELECT` вернёт один кортеж, несмотря на то, что в столбец `nick` записано значение `larry`, а в запросе фигурирует `Larry`.

F.9.3. Поведение при сравнении строк

Модуль `citext` выполняет сравнения, приводя каждую строку к нижнему регистру (как если бы вызывалась функция `lower`) и затем производя сравнения как обычно. Так, например, две строки будут считаться равными, если функция `lower`, обработав их, выдаст одинаковые результаты.

Чтобы имитировать правило сортировки без учёта регистра в максимально возможной степени, этот модуль предоставляет специальные, ориентированные на `citext`, операторы и функции для обработки строки. Так, например, операторы регулярных выражений `~` и `~*` действуют в том же ключе, когда применяются к типу `citext`: оба они не учитывают регистр. Это же распространяется на операторы `!~` и `!~*`, а также операторы `LIKE` `~~`, `~~*`, `!~~` и `!~~*`. Если же вы хотите, чтобы эти операторы учитывали регистр, вы можете привести их аргументы к типу `text`.

Подобным образом, все следующие функции выполняют сопоставления без учёта регистра, если их аргументы имеют тип `citext`:

- `regexp_match()`
- `regexp_matches()`
- `regexp_replace()`
- `regexp_split_to_array()`
- `regexp_split_to_table()`
- `replace()`

- `split_part()`
- `strpos()`
- `translate()`

Для функций с регулярными выражениями, если вам нужно регистрозависимое сопоставление, вы можете добавить флаг «c», чтобы принудительно включить этот режим. Чтобы получить регистрозависимое поведение без этого флага, вы должны привести аргумент к типу `text`, прежде чем вызывать эту функцию.

F.9.4. Ограничения

- Смена регистра символов в `citext` зависит от параметра `LC_CTYPE` вашей базы данных. Таким образом, как будут сравниваться значения, определяется при создании базы данных. На самом деле, по определениям стандарта Unicode, это сравнение не будет истинно регистронезависимым. По сути это означает, что если вас устраивает установленное правило сортировки, вас должны устраивать и сравнения `citext`. Но если в вашей базе данных хранятся строки на разных языках, пользователи одного языка могут получать неожиданные результаты запросов, если правило сортировки предназначено для другого языка.
- Начиная с PostgreSQL версии 9.1, вы можете добавлять указание `COLLATE` к значениям данных или столбцам `citext`. В настоящее время операторы `citext` принимают во внимание такое явное указание `COLLATE`, сравнивая строки в нижнем регистре, но изначальное приведение в нижний регистр всегда выполняется согласно параметру `LC_CTYPE` базы данных (как если бы указывалось `COLLATE "default"`). Это может быть изменено в будущем, чтобы на обоих этапах учитывалось указание `COLLATE` во входных данных.
- Тип `citext` не так эффективен, как `text`, так как функции операторов и функции сравнения для B-дерева должны делать копии данных и переводить их в нижний регистр для сравнения. Однако он несколько эффективнее варианта с применением `lower` для получения регистронезависимого сравнения.
- Тип `citext` малополезен в ситуациях, когда вам нужно сравнивать данные без учёта регистра в одних контекстах, и с учётом регистра — в других. Обычно в таких случаях используют `text` и вручную применяют функцию `lower`, когда нужно выполнить сравнение без учёта регистра; это прекрасно работает, если регистронезависимое сравнение требуется выполнять относительно редко. Если же почти всегда сравнение должно быть регистронезависимым и только иногда регистрозависимым, имеет смысл сохранить данные в столбце типа `citext`, и явно приводить их к типу `text` для регистрозависимого сравнения. В любом случае, чтобы оба варианта поиска были быстрыми, вам потребуются два индекса.
- Схема, содержащая операторы `citext`, должна находиться в текущем пути `search_path` (обычно это схема `public`); в противном случае будут вызываться регистрозависимые операторы для типа `text`.

F.9.5. Автор

Дэвид Е. Уилер `<david@kineticode.com>`

Разработку вдохновил оригинальный модуль `citext` Дональда Фрейзера.

F.10. spi

Модуль spi предоставляет несколько рабочих примеров использования Интерфейса программирования сервера (Server Programming Interface, SPI) и триггеров. Хотя эти функции имеют некоторую ценность сами по себе, они ещё более полезны как заготовки, которые можно приспособить под собственные нужды. Эти функции достаточно общие, чтобы работать с любой таблицей, но вы должны явно указать имена таблицы и полей (как описано ниже) при создании триггера.

Каждая группа функций, описанная ниже, представлена в виде отдельно устанавливаемого расширения.

F.10.1. refint — функции для реализации ссылочной целостности

Функции `check_primary_key()` и `check_foreign_key()` применяются для проверки ограничений внешних ключей. (Эта функциональность уже давно вытеснена встроенным механизмом внешних ключей, но этот модуль всё ещё полезен в качестве примера.)

Функция `check_primary_key()` проверяет ссылающуюся таблицу. Чтобы воспользоваться ей, создайте триггер `BEFORE INSERT OR UPDATE` с этой функцией для таблицы, ссылающейся на другую. Укажите в аргументах триггера: имена столбцов ссылающейся таблицы, образующих внешний ключ, имя целевой таблицы и имена столбцов в ней, образующих первичный/уникальный ключ. Чтобы контролировать несколько внешних ключей, создайте триггер для каждой такой ссылки.

Функция `check_foreign_key()` проверяет целевую таблицу. Чтобы использовать её, создайте триггер `BEFORE DELETE OR UPDATE` с этой функцией для таблицы, на которую ссылаются другие. Укажите в аргументах триггера: число ссылающихся таблиц, для которых функция должна выполнить проверки, действие в случае обнаружения ссылающегося ключа (`cascade` — удалить ссылающуюся строку, `restrict` — прервать транзакцию, `setnull` — установить в ссылающихся полях значения NULL), имена столбцов целевой таблицы, образующих первичный/уникальный ключ, а затем имена таблиц и столбцов (в количестве, задаваемом первым аргументом). Заметьте, что поля первичных/уникальных столбцов должны иметь пометку NOT NULL и по ним должен быть создан уникальный индекс.

Примеры приведены в `refint.example`.

F.10.2. timetravel — функции для реализации перемещений во времени

В далёком прошлом в Postgres Pro была встроенная возможность перемещений во времени, для которой фиксировалось время добавления и удаления каждого кортежа. Эти функции позволяют её имитировать. Чтобы использовать их, вы должны добавить в таблицу два столбца типа `abstime`, в которых будет храниться дата/время, когда кортеж был вставлен (start_date) и когда изменён/удалён (stop_date):

```
CREATE TABLE mytab (
        ...             ...
        start_date      abstime,
        stop_date       abstime
        ...             ...
);
```

Эти столбцы могут называться как угодно, но в данном описании они называются start_date и stop_date.

Когда вставляется новая строка, в start_date обычно устанавливается текущее время, а в stop_date — `infinity` (бесконечность). Триггер автоматически подставит эти значения, если добавляемая строка содержит NULL в этих столбцах. Обычно не NULL в этих столбцах может оказаться только при загрузке в базу выгруженных данных.

Кортежи, в которых поле stop_date равно `infinity`, считаются «актуальными сейчас» и могут быть изменены. Кортежи с определённой датой stop_date больше не могут быть изменены — триггер будет препятствовать этому. (Если вам нужно сделать это, вы можете отключить машину времени как показано ниже.)

Если кортеж является изменяемым, при модификации в нём меняется только stop_date (на текущее время), но в таблицу вставляется новый кортеж с модифицированными данными. В поле start_date в этом новом кортеже записывается текущее время, а в stop_date записывается `infinity`.

При удалении кортеж на самом деле не удаляется; в нём только записывается текущее время в stop_date.

Чтобы запросить кортежи «актуальные сейчас», добавьте `stop_date = 'infinity'` в условие WHERE вашего запроса. (Возможно, вы захотите завернуть это условие в представление.) Аналогичным образом вы можете запрашивать кортежи, которые были актуальны в любой момент в прошлом, задав подходящие условия для start_date и stop_date.

Функция `timetravel()` реализует код универсального триггера, поддерживающего это поведение. Чтобы использовать её, создайте триггер `BEFORE INSERT OR UPDATE OR DELETE` с этой функцией для каждой таблицы, перемещающейся во времени. Передайте триггеру два аргумента: фактические имена столбцов start_date и stop_date. Вы также можете дополнительно передать от одного до трёх аргументов, задающих имена столбцов типа `text`. Данный триггер сохранит имя текущего пользователя в первый из этих столбцов при INSERT, во второй — при UPDATE, и в третий — при DELETE.

Функция `set_timetravel()` позволяет включить или отключить машину времени для таблицы. Вызов `set_timetravel('mytab', 1)` включает машину времени для таблицы `mytab`, а `set_timetravel('mytab', 0)` — отключает её для таблицы `mytab`. В обоих случаях возвращается прежнее состояние. Когда машина времени выключена, вы можете свободно модифицировать столбцы start_date и stop_date. Заметьте, что состояние активности машины является локальным для текущего сеанса базы данных — в новых сеансах машина времени всегда включена для всех таблиц.

Функция `get_timetravel()` возвращает состояние перемещения во времени для таблицы, не меняя его.

Пример приведён в `timetravel.example`.

F.10.3. autoinc — функции для автоувеличения полей

Функция `autoinc()` реализует код триггера, сохраняющего следующее значение последовательности в целочисленном поле. Это в некоторой степени пересекается со встроенной функциональностью столбца «serial», но есть и отличия: `autoinc()` препятствует попыткам вставить другое значение поля при добавлении строк и может увеличивать значение поля при изменениях.

Чтобы использовать её, создайте триггер `BEFORE INSERT` (или `BEFORE INSERT OR UPDATE`) с этой функцией. Передайте триггеру два аргумента: имя целочисленного столбца, который будет меняться, и имя объекта последовательности, который будет поставлять значения. (Вообще вы можете задать любое число пар таких имён, если хотите поддерживать несколько автоувеличивающихся столбцов.)

Пример приведён в `autoinc.example`.

F.10.4. insert_username — функции для отслеживания пользователя, вносящего изменения

Функция `insert_username()` реализует код триггера, сохраняющего имя текущего пользователя в текстовом поле. Это может быть полезно для отслеживания пользователя, изменившего конкретную строку таблицы последним.

Чтобы использовать её, создайте триггер `BEFORE INSERT` и/или `UPDATE` с этой функцией. Передайте триггеру один аргумент: имя целевого текстового столбца.

Пример приведён в `insert_username.example`.

F.10.5. moddatetime — функции для отслеживания времени последнего изменения

Функция `moddatetime()` реализует код триггера, сохраняющего текущее время в поле типа `timestamp`. Это может быть полезно для отслеживания времени последней модификации конкретной строки таблицы.

Чтобы использовать её, создайте триггер `BEFORE UPDATE` с этой функцией. Передайте триггеру один аргумент: имя целевого столбца. Столбец должен иметь тип `timestamp` или `timestamp with time zone`.

Пример приведён в `moddatetime.example`.

F.11. cube

Этот модуль реализует тип данных `cube` для представления многомерных кубов.

F.11.1. Синтаксис

В Таблице F.3 показаны внешние представления типа `cube`. Буквы *x*, *y* и т. д. обозначают числа с плавающей точкой.

Таблица F.3. Внешние представления кубов

Внешний синтаксис	Значение
x	Одномерная точка (или одномерный интервал нулевой длины)
(*x*)	То же, что и выше
x1, *x2*,..., *xn*	Точка в n-мерном пространстве, представленная внутри как куб нулевого объёма
(*x1*, *x2*,..., *xn*)	То же, что и выше
(*x*),(*y*)	Одномерный интервал, начинающийся в точке *x* и заканчивающийся в *y*, либо наоборот; порядок значения не имеет
[(*x*),(*y*)]	То же, что и выше
(*x1*,..., *xn*),(*y1*,..., *yn*)	N-мерный куб, представленный парой диагонально противоположных углов
[(*x1*,..., *xn*),(*y1*,..., *yn*)]	То же, что и выше

В каком порядке вводятся противоположные углы куба, не имеет значения. Функции, принимающие тип `cube`, автоматически меняют углы местами, чтобы получить единое внутреннее представление «левый нижний — правый верхний». Когда эти углы совмещаются, в `cube` для экономии пространства хранится только один угол с флагом «является точкой».

Пробельные символы игнорируются, так что `[(`*x*`),(`*y*`)]` не отличается от `[ ( `*x*` ), ( `*y*` ) ]`.

F.11.2. Точность

Значения хранятся внутри как 64-битные числа с плавающей точкой. Это значит, что числа с более чем 16 значащими цифрами будут усекаться.

F.11.3. Использование

В Таблице F.4 показаны операторы, предназначенные для работы с типом `cube`.

Таблица F.4. Операторы для кубов

Оператор	Результат	Описание
`a = b`	`boolean`	Кубы a и b идентичны.
`a && b`	`boolean`	Кубы a и b пересекаются.
`a @> b`	`boolean`	Куб a включает куб b.

Оператор	**Результат**	**Описание**
a <@ b	boolean	Куб a включён в куб b.
a < b	boolean	Куб a меньше куба b.
a <= b	boolean	Куб a меньше или равен кубу b.
a > b	boolean	Куб a больше куба b.
a >= b	boolean	Куб a больше или равен кубу b.
a <> b	boolean	Куб a не равен кубу b.
a -> n	float8	Выдаёт *n*-ную координату куба (считая с 1).
a ~> n	float8	Выдаёт *n*-ную координату куба следующим образом: n = 2 * k - 1 обозначает нижнюю границу *k*-ой размерности, n = 2 * k обозначает верхнюю границу *k*-ой размерности. Этот оператор предназначен для поддержки KNN-GiST.
a <-> b	float8	Евклидово расстояние между a и b.
a <#> b	float8	Расстояние городских кварталов (метрика L-1) между a и b.
a <=> b	float8	Расстояние Чебышева (метрика L-inf) между a и b.

(До версии PostgreSQL 8.2 операторы включения @> и <@ обозначались соответственно как @ и ~. Эти имена по-прежнему действуют, но считаются устаревшими и в конце концов будут упразднены. Заметьте, что старые имена произошли из соглашения, которому раньше следовали ключевые геометрические типы данных!)

Скалярные операторы упорядочивания (<, >= и т. д.) не имеют большого смысла ни для каких практических целей, кроме сортировки. Эти операторы сначала сравнивают первые координаты и если они равны, сравнивают вторые и т. д. Они предназначены в основном для поддержки класса операторов индекса-B-дерева для типа cube, который может быть полезен, например, если вы хотите создать ограничение UNIQUE для столбца типа cube.

Модуль cube также предоставляет класс операторов индекса GiST для значений cube. Индекс GiST для cube может применяться для поиска значений в выражениях с операторами =, &&, @> и <@ в предложениях WHERE.

GiST-индекс для cube может быть полезен и для поиска ближайших соседей с использованием операторов метрики <->, <#> и <=> в предложениях ORDER BY. Например, ближайшего соседа точки в трёхмерном пространстве (0.5, 0.5, 0.5) можно эффективно найти так:

```
SELECT c FROM test ORDER BY c <-> cube(array[0.5,0.5,0.5]) LIMIT 1;
```

Оператор ~> может также использоваться таким образом, чтобы эффективно выдавать первые несколько значений, отсортированных по выбранной координате. Например, чтобы получить первые несколько кубов, упорядоченных по возрастанию первой координаты (левого нижнего угла), можно использовать следующий запрос:

```
SELECT c FROM test ORDER BY c ~> 1 LIMIT 5;
```

А чтобы получить двумерные кубы, отсортированные по убыванию первой координаты правого верхнего угла:

```
SELECT c FROM test ORDER BY c ~> 3 DESC LIMIT 5;
```

В Таблице F.5 перечислены все доступные функции.

Таблица F.5. Функции для работы с кубами

Функция	Результат	Описание	Пример
`cube(float8)`	`cube`	Создаёт одномерный куб, у которого обе координаты равны.	`cube(1) == '(1)'`
`cube(float8, float8)`	`cube`	Создаёт одномерный куб.	`cube(1,2) == '(1),(2)'`
`cube(float8[])`	`cube`	Создаёт куб нулевого объёма по координатам, определяемым массивом.	`cube(ARRAY[1,2]) == '(1,2)'`
`cube(float8[], float8[])`	`cube`	Создаёт куб с координатами правого верхнего и левого нижнего углов, определяемыми двумя массивами, которые должны быть одинаковой длины.	`cube(ARRAY[1,2], ARRAY[3,4]) == '(1,2),(3,4)'`
`cube(cube, float8)`	`cube`	Создаёт новый куб, добавляя размерность к существующему кубу с одинаковым значением новой координаты для обеих углов. Это бывает полезно, когда нужно построить кубы поэтапно из вычисляемых значений.	`cube('(1,2),(3,4)'::cube, 5) == '(1,2,5),(3,4,5)'`
`cube(cube, float8, float8)`	`cube`	Создаёт новый куб, добавляя размерность к существующему кубу. Это бывает полезно, когда нужно построить кубы поэтапно из вычисляемых значений.	`cube('(1,2),(3,4)'::cube, 5, 6) == '(1,2,5),(3,4,6)'`
`cube_dim(cube)`	`integer`	Возвращает число размерностей куба.	`cube_dim('(1,2),(3,4)') == '2'`
`cube_ll_coord(cube, integer)`	`float8`	Возвращает значение *n*-ной координаты левого нижнего угла куба.	`cube_ll_coord('(1,2),(3,4)', 2) == '2'`
`cube_ur_coord(cube, integer)`	`float8`	Возвращает значение *n*-ной координаты правого верхнего угла куба.	`cube_ur_coord('(1,2),(3,4)', 2) == '4'`

Функция	**Результат**	**Описание**	**Пример**
`cube_is_point( cube)`	`boolean`	Возвращает true, если куб является точкой, то есть если два определяющих его угла совпадают.	
`cube_distance(cube, cube)`	`float8`	Возвращает расстояние между двумя кубами. Если оба куба являются точками, вычисляется обычная функция расстояния.	
`cube_subset(cube, integer[])`	`cube`	Создаёт новый куб из существующего, используя список размерностей из массива. Может применяться для получения координат углов в одном измерении, для удаления измерений и изменения их порядка.	`cube_subset(cube( '(1,3,5),(6,7, 8)'), ARRAY[2]) == '(3),(7)' cube_ subset(cube('(1, 3,5),(6,7,8)'), ARRAY[3,2,1,1]) == '(5,3,1,1),( 8,7,6,6)'`
`cube_union(cube, cube)`	`cube`	Создаёт объединение двух кубов.	
`cube_inter(cube, cube)`	`cube`	Создаёт пересечение двух кубов.	
`cube_enlarge(c cube, r double, n integer)`	`cube`	Увеличивает размер куба на заданный радиус *r* как минимум в *n* измерениях. Если радиус отрицательный, куб, наоборот, уменьшается. Все определённые измерения изменяются на величину радиуса *r*. Координаты левого нижнего угла уменьшаются на *r*, а координаты правого верхнего увеличиваются на *r*. Если координата левого нижнего угла становится больше соответствующей координаты правого верхнего (это возможно, только когда *r* < 0), обоим координатам присваивается их среднее значение. Если *n* превышает	`cube_enlarge('(1, 2),(3,4)', 0.5, 3) == '(0.5,1.5, -0.5),(3.5,4.5, 0.5)'`

Функция	Результат	Описание	Пример
		число определённых измерений и куб увеличивается ($r > 0$), добавляются дополнительные размерности, недостающие до n; начальным значением для дополнительных координат считается ноль. Эта функция полезна для создания окружающих точку прямоугольников для поиска ближайших точек.	

F.11.4. Поведение по умолчанию

Я полагаю, что это объединение:

```
select cube_union('(0,5,2),(2,3,1)', '0');
cube_union
-------------------
(0, 0, 0),(2, 5, 2)
(1 row)
```

не противоречит здравому смыслу, как и это пересечение

```
select cube_inter('(0,-1),(1,1)', '(-2),(2)');
cube_inter
-------------
(0, 0),(1, 0)
(1 row)
```

Во всех бинарных операциях с кубами разных размерностей, я полагаю, что куб с меньшей размерностью является декартовой проекцией; то есть в опущенных в строковом представлении координатах предполагаются нули. Таким образом, показанные выше вызовы равнозначны следующим:

```
cube_union('(0,5,2),(2,3,1)','(0,0,0),(0,0,0)');
cube_inter('(0,-1),(1,1)','(-2,0),(2,0)');
```

В следующем предикате включения применяется синтаксис точек, хотя фактически второй аргумент представляется внутри кубом. Этот синтаксис избавляет от необходимости определять отдельный тип точек и функции для предикатов (cube,point).

```
select cube_contains('(0,0),(1,1)', '0.5,0.5');
cube_contains
--------------
t
(1 row)
```

F.11.5. Замечания

Примеры использования можно увидеть в регрессионном тесте `sql/cube.sql`.

Во избежание некорректного применения этого типа, число размерностей кубов искусственно ограничено значением 100. Если это ограничение вас не устраивает, его можно изменить в `cubedata.h`.

F.11.6. Благодарности

Первый автор: Джин Селков мл. <selkovjr@mcs.anl.gov>, Аргоннская национальная лаборатория, Отдел математики и компьютерных наук

Я очень благодарен в первую очередь профессору Джо Геллерштейну (*https://dsf.berkeley.edu/jmh/*) за пояснение сути GiST (*http://gist.cs.berkeley.edu/*) и его бывшему студенту, Энди Донгу, за пример, написанный для Illustra. Я также признателен всем разработчикам Postgres в настоящем и прошлом за возможность создать свой собственный мир и спокойно жить в нём. Ещё я хотел бы выразить признательность Аргоннской лаборатории и Министерству энергетики США за годы постоянной поддержки моих исследований в области баз данных.

Небольшие изменения в этот пакет внёс Бруно Вольф III <bruno@wolff.to> в августе/сентябре 2002 г. В том числе он перешёл от одинарной к двойной точности и добавил несколько новых функций.

Дополнительные изменения внёс Джошуа Рейх <josh@root.net> в июле 2006 г. В частности, он добавил `cube(float8[], float8[])`, подчистил код и перевёл его на протокол вызовов версии V1 с устаревшего протокола V0.

F.12. dblink

Модуль `dblink` обеспечивает подключения к другим базам данных Postgres Pro из сеанса базы данных.

См. также описание модуля postgres_fdw, который предоставляет примерно ту же функциональность, но через более современную и стандартизированную инфраструктуру.

dblink_connect

dblink_connect — открывает постоянное подключение к удалённой базе данных

Синтаксис

```
dblink_connect(text connstr) returns text
dblink_connect(text connname, text connstr) returns text
```

Описание

Функция `dblink_connect()` устанавливает подключение к удалённой базе данных Postgres Pro. Целевой сервер и база данных указываются в стандартной строке подключения libpq. Если требуется, этому подключению можно назначить имя. В один момент времени могут быть открытыми несколько именованных подключений, но только одно подключение без имени. Подключение будет сохраняться, пока не будет закрыто или до завершения сеанса базы данных.

В строке подключения также может задаваться имя существующего стороннего сервера. Для определения стороннего сервера рекомендуется использовать обёртку сторонних данных `dblink_fdw`. См. пример ниже, а также CREATE SERVER и CREATE USER MAPPING.

Аргументы

connname

Имя, назначаемое этому подключению; если опускается, открывается безымянное подключение, заменяющее ранее существующее безымянное подключение.

connstr

Строка подключения в стиле libpq, например `hostaddr=127.0.0.1 port=5432 dbname=mydb user=postgres password=mypasswd options=-csearch_path=`. За подробностями обратитесь к Подразделу 31.1.1. В ней также может задаваться имя стороннего сервера.

Возвращаемое значение

Возвращает состояние (это всегда строка `OK`, так как в случае любой ошибки функция прерывается, выдавая исключение).

Замечания

Если к базе данных, которая не приведена в соответствие шаблону безопасного использования схем, имеют доступ недоверенные пользователи, начинайте сеанс с удаления доступных им для записи схем из пути поиска (`search_path`). Например, для этого можно добавить `options=-csearch_path=` в *connstr*. Это касается не только `dblink`, но и любых других интерфейсов для выполнения произвольных SQL-команд.

Создавать подключения, не требующие аутентификации по паролю, с помощью `dblink_connect` разрешено только суперпользователям. Если эта возможность нужна обычным пользователям, следует воспользоваться функцией `dblink_connect_u`.

Использовать в именах подключений знаки «равно» не рекомендуется, так как при этом возможна путаница со строками подключений в других функциях `dblink`.

Примеры

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_connect('myconn', 'dbname=postgres options=-csearch_path=');
```

```
 dblink_connect
----------------
 OK
(1 row)

-- Функциональность обёртки сторонних данных (FOREIGN DATA WRAPPER)
-- Замечание: чтобы это работало, для локальных подключений требуется аутентификация по
 паролю
--       В противном случае, вызвав dblink_connect(), вы получите:
--       ------------------------------------------------------------------------
--       ERROR:  password is required
--       DETAIL:  Non-superuser cannot connect if the server does not request a
 password.
--       HINT:  Target server's authentication method must be changed.
--
--       ОШИБКА:  требуется пароль
--       ПОДРОБНОСТИ:  Обычный пользователь не может подключиться, если сервер не
 требует пароль.
--       ПОДСКАЗКА:  Необходимо изменить метод аутентификации целевого сервера.

CREATE SERVER fdtest FOREIGN DATA WRAPPER dblink_fdw OPTIONS (hostaddr '127.0.0.1',
 dbname 'contrib_regression');

CREATE USER regress_dblink_user WITH PASSWORD 'secret';
CREATE USER MAPPING FOR regress_dblink_user SERVER fdtest OPTIONS (user
 'regress_dblink_user', password 'secret');
GRANT USAGE ON FOREIGN SERVER fdtest TO regress_dblink_user;
GRANT SELECT ON TABLE foo TO regress_dblink_user;

\set ORIGINAL_USER :USER
\c - regress_dblink_user
SELECT dblink_connect('myconn', 'fdtest');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('myconn', 'SELECT * FROM foo') AS t(a int, b text, c text[]);
 a  | b |       c
----+---+---------------
  0 | a | {a0,b0,c0}
  1 | b | {a1,b1,c1}
  2 | c | {a2,b2,c2}
  3 | d | {a3,b3,c3}
  4 | e | {a4,b4,c4}
  5 | f | {a5,b5,c5}
  6 | g | {a6,b6,c6}
  7 | h | {a7,b7,c7}
  8 | i | {a8,b8,c8}
  9 | j | {a9,b9,c9}
 10 | k | {a10,b10,c10}
(11 rows)

\c - :ORIGINAL_USER
REVOKE USAGE ON FOREIGN SERVER fdtest FROM regress_dblink_user;
REVOKE SELECT ON TABLE foo FROM regress_dblink_user;
DROP USER MAPPING FOR regress_dblink_user SERVER fdtest;
DROP USER regress_dblink_user;
```

```
DROP SERVER fdtest;
```

dblink_connect_u

dblink_connect_u — открывает постоянное подключение к удалённой базе данных, небезопасно

Синтаксис

```
dblink_connect_u(text connstr) returns text
dblink_connect_u(text connname, text connstr) returns text
```

Описание

Функция `dblink_connect_u()` не отличается от `dblink_connect()`, за исключением того, что она позволяет подключаться с любым методом аутентификации обычным пользователям.

Если удалённый сервер выбирает режим аутентификации без пароля, возможно олицетворение и последующее повышение привилегий, так как сеанс будет установлен от имени пользователя, который исполняет локальный процесс Postgres Pro. Кроме того, даже если удалённый сервер запрашивает пароль, этот пароль можно получить из среды сервера, например, из файла `~/.pgpass`, принадлежащего пользователю сервера. Это чревато не только олицетворением, но и выдачей пароля не заслуживающему доверия удалённому серверу. Поэтому `dblink_connect_u()` изначально устанавливается так, что роль `PUBLIC` лишена всех прав на её использование, то есть вызывать её могут только суперпользователи. В некоторых ситуациях допустимо дать право `EXECUTE` для `dblink_connect_u()` определённым пользователям, которым можно доверять, но это нужно делать осторожно. Также рекомендуется убедиться в том, что файл `~/.pgpass`, принадлежащий пользователю сервера, *не* содержит никаких записей со звёздочкой в качестве имени узла.

За дополнительными подробностями обратитесь к описанию `dblink_connect()`.

dblink_disconnect

dblink_disconnect — закрывает постоянное подключение к удалённой базе данных

Синтаксис

```
dblink_disconnect() returns text
dblink_disconnect(text connname) returns text
```

Описание

`dblink_disconnect()` закрывает подключение, ранее открытое функцией `dblink_connect()`. Форма без аргументов закрывает безымянное подключение.

Аргументы

connname

Имя закрываемого именованного подключения.

Возвращаемое значение

Возвращает состояние (это всегда строка `OK`, так как в случае любой ошибки функция прерывается, выдавая исключение).

Примеры

```
SELECT dblink_disconnect();
 dblink_disconnect
-------------------
 OK
(1 row)

SELECT dblink_disconnect('myconn');
 dblink_disconnect
-------------------
 OK
(1 row)
```

dblink

dblink — выполняет запрос в удалённой базе данных

Синтаксис

```
dblink(text connname, text sql [, bool fail_on_error]) returns setof record
dblink(text connstr, text sql [, bool fail_on_error]) returns setof record
dblink(text sql [, bool fail_on_error]) returns setof record
```

Описание

`dblink` выполняет запрос (обычно `SELECT`, но это может быть и любой другой оператор SQL, возвращающий строки) в удалённой базе данных.

Когда этой функции передаются два аргумента типа `text`, первый сначала рассматривается как имя постоянного подключения; если такое подключение находится, команда выполняется для него. Если не находится, первый аргумент воспринимается как строка подключения, как для функции `dblink_connect`, и заданное подключение устанавливается только на время выполнения этой команды.

Аргументы

connname

Имя используемого подключения; опустите этот параметр, чтобы использовать безымянное подключение.

connstr

Строка подключения, описанная ранее для `dblink_connect`

sql

SQL-запрос, который вы хотите выполнить в удалённой базе данных, например `select * from foo`.

fail_on_error

Если равен true (это значение по умолчанию), в случае ошибки, выданной на удалённой стороне соединения, ошибка также выдаётся локально. Если равен false, удалённая ошибка выдаётся локально как ЗАМЕЧАНИЕ, и функция не возвращает строки.

Возвращаемое значение

Эта функция возвращает строки, выдаваемые в результате запроса. Так как `dblink` может выполнять произвольные запросы, она объявлена как возвращающая тип `record`, а не некоторый определённый набор столбцов. Это означает, что вы должны указать ожидаемый набор столбцов в вызывающем запросе — в противном случае Postgres Pro не будет знать, чего ожидать. Например:

```
SELECT *
    FROM dblink('dbname=mydb options=-csearch_path=',
                'select proname, prosrc from pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

В части «псевдонима» предложения `FROM` должны указываться имена столбцов и типы, которые будет возвращать функция. (Указание имён столбцов в псевдониме таблицы предусмотрено стандартом SQL, но определение типов столбцов является расширением Postgres Pro.) Это позволяет системе понять, во что должно разворачиваться обозначение `*`, и на что ссылается `proname` в предложении `WHERE`, прежде чем пытаться выполнять эту функцию. Во время выполнения произойдёт ошибка, если действительный результат запроса из удалённой базы данных не будет

содержать столько столбцов, сколько указано в предложении FROM. Однако имена столбцов могут не совпадать, так же, как dblink не настаивает на точном совпадении типов. Функция завершится успешно, если возвращаемые строки данных будут допустимыми для ввода в тип столбца, объявленный в предложении FROM.

Замечания

Использовать dblink с предопределёнными запросами будет удобнее, если создать представление. Это позволит скрыть в его определении информацию о типах столбцов и не выписывать её в каждом запросе. Например:

```
CREATE VIEW myremote_pg_proc AS
  SELECT *
    FROM dblink('dbname=postgres options=-csearch_path=',
                'select proname, prosrc from pg_proc')
    AS t1(proname name, prosrc text);

SELECT * FROM myremote_pg_proc WHERE proname LIKE 'bytea%';
```

Примеры

```
SELECT * FROM dblink('dbname=postgres options=-csearch_path=',
                     'select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteain    | byteain
 byteaout   | byteaout
(12 rows)

SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
```

```
 byteain    | byteain
 byteaout   | byteaout
(12 rows)

SELECT dblink_connect('myconn', 'dbname=regression options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('myconn', 'select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 bytearecv  | bytearecv
 byteasend  | byteasend
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteain    | byteain
 byteaout   | byteaout
(14 rows)
```

dblink_exec

dblink_exec — выполняет команду в удалённой базе данных

Синтаксис

```
dblink_exec(text connname, text sql [, bool fail_on_error]) returns text
dblink_exec(text connstr, text sql [, bool fail_on_error]) returns text
dblink_exec(text sql [, bool fail_on_error]) returns text
```

Описание

Функция `dblink_exec` выполняет команду (то есть любой SQL-оператор, не возвращающий строки) в удалённой базе данных.

Когда этой функции передаются два аргумента типа `text`, первый сначала рассматривается как имя постоянного подключения; если такое подключение находится, команда выполняется для него. Если не находится, первый аргумент воспринимается как строка подключения, как для функции `dblink_connect`, и заданное подключение устанавливается только на время выполнения этой команды.

Аргументы

connname

Имя используемого подключения; опустите этот параметр, чтобы использовать безымянное подключение.

connstr

Строка подключения, описанная ранее для `dblink_connect`

sql

SQL-запрос, который вы хотите выполнить в удалённой базе данных, например `insert into foo values(0, 'a', '{"a0","b0","c0"}')`.

fail_on_error

Если равен true (это значение по умолчанию), в случае ошибки, выданной на удалённой стороне соединения, ошибка также выдаётся локально. Если равен false, удалённая ошибка выдаётся локально как ЗАМЕЧАНИЕ, и возвращаемым значением функции будет `ERROR`.

Возвращаемое значение

Возвращает состояние (либо строку состояния команды, либо `ERROR`).

Примеры

```
SELECT dblink_connect('dbname=dblink_test_standby');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_exec('insert into foo values(21, ''z'', ''{"a0","b0","c0"}'');');
   dblink_exec
-----------------
 INSERT 943366 1
(1 row)
```

```
SELECT dblink_connect('myconn', 'dbname=regression');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_exec('myconn', 'insert into foo values(21, ''z'',
 ''{"a0","b0","c0"}'');');
   dblink_exec
------------------
 INSERT 6432584 1
(1 row)

SELECT dblink_exec('myconn', 'insert into pg_class values (''foo'')',false);
NOTICE:  sql error
DETAIL:  ERROR:  null value in column "relnamespace" violates not-null constraint

 dblink_exec
-------------
 ERROR
(1 row)
```

dblink_open

dblink_open — открывает курсор в удалённой базе данных

Синтаксис

```
dblink_open(text cursorname, text sql [, bool fail_on_error]) returns text
dblink_open(text connname, text cursorname, text sql [, bool fail_on_error]) returns
 text
```

Описание

Функция `dblink_open()` открывает курсор в удалённой базе данных. Открытым курсором можно будет манипулировать функциями `dblink_fetch()` и `dblink_close()`.

Аргументы

connname

Имя используемого подключения; опустите этот параметр, чтобы использовать безымянное подключение.

cursorname

Имя, назначаемое курсору.

sql

Оператор `SELECT`, который вы хотите выполнять в удалённой базе данных, например `select * from pg_class`.

fail_on_error

Если равен true (это значение по умолчанию), в случае ошибки, выданной на удалённой стороне соединения, ошибка также выдаётся локально. Если равен false, удалённая ошибка выдаётся локально как ЗАМЕЧАНИЕ, и возвращаемым значением функции будет `ERROR`.

Возвращаемое значение

Возвращает состояние, `OK` или `ERROR`.

Замечания

Так как курсор может существовать только в рамках транзакции, функция `dblink_open` начинает явный блок транзакции (командой `BEGIN`) на удалённой стороне, если транзакция там ещё не открыта. Эта транзакция будет снова закрыта при соответствующем вызове `dblink_close`. Заметьте, что если вы с помощью `dblink_exec` изменяете данные между вызовами `dblink_open` и `dblink_close`, а затем происходит ошибка, либо если вы вызываете `dblink_disconnect` перед `dblink_close`, ваши изменения *будут потеряны*, так как транзакция будет прервана.

Примеры

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-------------
```

```
 OK
(1 row)
```

dblink_fetch

dblink_fetch — возвращает строки из открытого курсора в удалённой базе данных

Синтаксис

```
dblink_fetch(text cursorname, int howmany [, bool fail_on_error]) returns setof record
dblink_fetch(text connname, text cursorname, int howmany [, bool fail_on_error])
 returns setof record
```

Описание

`dblink_fetch` выбирает строки из курсора, ранее открытого функцией `dblink_open`.

Аргументы

connname

Имя используемого подключения; опустите этот параметр, чтобы использовать безымянное подключение.

cursorname

Имя курсора, из которого выбираются данные.

howmany

Максимальное число строк, которое нужно получить. Данная функция выбирает через курсор следующие *howmany* строк, начиная с текущей позиции курсора и двигаясь вперёд. Когда курсор доходит до конца, строки больше не выдаются.

fail_on_error

Если равен true (это значение по умолчанию), в случае ошибки, выданной на удалённой стороне соединения, ошибка также выдаётся локально. Если равен false, удалённая ошибка выдаётся локально как ЗАМЕЧАНИЕ, и функция не возвращает строки.

Возвращаемое значение

Эта функция возвращает строки, выбираемые через курсор. Для использования этой функции необходимо задать ожидаемый набор столбцов, как ранее говорилось в описании `dblink`.

Замечания

При несовпадении числа возвращаемых столбцов, определённого в предложении `FROM`, с фактическим числом столбцов, возвращённых удалённым курсором, выдаётся ошибка. В этом случае удалённый курсор всё равно продвигается на столько строк, на сколько он продвинулся бы, если бы ошибка не произошла. То же самое верно для любых других ошибок, происходящих при локальной обработке результатов после выполнения удалённой команды `FETCH`.

Примеры

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc where proname like
 ''bytea%''');
 dblink_open
-------------
```

```
 OK
(1 row)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname |  source
----------+----------
 byteacat | byteacat
 byteacmp | byteacmp
 byteaeq  | byteaeq
 byteage  | byteage
 byteagt  | byteagt
(5 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname  |  source
-----------+-----------
 byteain   | byteain
 byteale   | byteale
 bytealike | bytealike
 bytealt   | bytealt
 byteane   | byteane
(5 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
  funcname  |   source
------------+------------
 byteanlike | byteanlike
 byteaout   | byteaout
(2 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname | source
----------+--------
(0 rows)
```

dblink_close

dblink_close — закрывает курсор в текущей базе данных

Синтаксис

```
dblink_close(text cursorname [, bool fail_on_error]) returns text
dblink_close(text connname, text cursorname [, bool fail_on_error]) returns text
```

Описание

`dblink_close` закрывает курсор, ранее открытый функцией `dblink_open`.

Аргументы

connname

Имя используемого подключения; опустите этот параметр, чтобы использовать безымянное подключение.

cursorname

Имя курсора, который будет закрыт.

fail_on_error

Если равен true (это значение по умолчанию), в случае ошибки, выданной на удалённой стороне соединения, ошибка также выдаётся локально. Если равен false, удалённая ошибка выдаётся локально как ЗАМЕЧАНИЕ, и возвращаемым значением функции будет `ERROR`.

Возвращаемое значение

Возвращает состояние, `OK` или `ERROR`.

Замечания

Если вызов `dblink_open` начал явный блок транзакции и это последний открытый курсор, остававшийся в этом подключении, то `dblink_close` выполнит соответствующую команду `COMMIT`.

Примеры

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-------------
 OK
(1 row)

SELECT dblink_close('foo');
 dblink_close
--------------
 OK
(1 row)
```

dblink_get_connections

dblink_get_connections — возвращает имена всех открытых именованных подключений dblink

Синтаксис

```
dblink_get_connections() returns text[]
```

Описание

`dblink_get_connections` возвращает массив имён всех открытых именованных подключений `dblink`.

Возвращаемое значение

Возвращает текстовый массив имён подключений, либо NULL, если они отсутствуют.

Примеры

```
SELECT dblink_get_connections();
```

dblink_error_message

dblink_error_message — выдаёт сообщение последней ошибки для именованного подключения

Синтаксис

```
dblink_error_message(text connname) returns text
```

Описание

`dblink_error_message` извлекает самое последнее сообщение удалённой ошибки для заданного подключения.

Аргументы

connname

Имя используемого подключения.

Возвращаемое значение

Возвращает сообщение последней ошибки либо `OK`, если в сеансе этого подключения не было ошибок.

Замечания

Когда функцией `dblink_send_query` передаются асинхронные запросы, сообщение об ошибке, связанное с текущим подключением, может не измениться до получения от сервера ответного сообщения. Поэтому обычно до вызова `dblink_error_message` следует вызывать функцию `dblink_is_busy` или `dblink_get_result`, чтобы ошибка, возникшая при выполнении асинхронного запроса, не осталась незамеченной.

Примеры

```
SELECT dblink_error_message('dtest1');
```

dblink_send_query

dblink_send_query — передаёт асинхронный запрос в удалённую базу данных

Синтаксис

```
dblink_send_query(text connname, text sql) returns int
```

Описание

`dblink_send_query` передаёт запрос для асинхронного выполнения, то есть не дожидается получения результата. С этим подключением не должен быть связан уже выполняющийся асинхронный запрос.

После успешной передачи асинхронного запроса состояние его завершения можно проверять, вызывая функцию `dblink_is_busy`, и в итоге получать данные, вызвав `dblink_get_result`. Также можно попытаться отменить активный асинхронный запрос, вызвав `dblink_cancel_query`.

Аргументы

connname

Имя используемого подключения.

sql

Оператор SQL, который вы хотите выполнить в удалённой базе данных, например `select * from pg_class`.

Возвращаемое значение

Возвращает 1, если запрос был успешно отправлен на обработку, или 0 в противном случае.

Примеры

```
SELECT dblink_send_query('dtest1', 'SELECT * FROM foo WHERE f1 < 3');
```

dblink_is_busy

dblink_is_busy — проверяет, не выполняется ли через подключение асинхронный запрос

Синтаксис

```
dblink_is_busy(text connname) returns int
```

Описание

`dblink_is_busy` проверяет, не выполняется ли асинхронный запрос.

Аргументы

connname

Имя проверяемого подключения.

Возвращаемое значение

Возвращает 1, если подключение занято, или 0 в противном случае. Если эта функция возвращает 0, гарантируется, что вызов `dblink_get_result` не будет заблокирован.

Примеры

```
SELECT dblink_is_busy('dtest1');
```

dblink_get_notify

dblink_get_notify — выдаёт асинхронные уведомления подключения

Синтаксис

```
dblink_get_notify() returns setof (notify_name text, be_pid int, extra text)
dblink_get_notify(text connname) returns setof (notify_name text, be_pid int, extra
 text)
```

Описание

`dblink_get_notify` выдаёт уведомления либо безымянного подключения, либо подключения с заданным именем. Чтобы получать уведомления через dblink, необходимо сначала выполнить `LISTEN`, воспользовавшись функцией `dblink_exec`. За подробностями обратитесь к LISTEN и NOTIFY.

Аргументы

connname

Имя именованного подключения, уведомления которого нужно получить.

Возвращаемое значение

Возвращает `setof (notify_name text, be_pid int, extra text)` или пустой набор, если уведомлений нет.

Примеры

```
SELECT dblink_exec('LISTEN virtual');
 dblink_exec
-------------
 LISTEN
(1 row)

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-------------+--------+-------
(0 rows)

NOTIFY virtual;
NOTIFY

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-------------+--------+-------
 virtual     |   1229 |
(1 row)
```

dblink_get_result

dblink_get_result — получает результат асинхронного запроса

Синтаксис

```
dblink_get_result(text connname [, bool fail_on_error]) returns setof record
```

Описание

`dblink_get_result` получает результаты асинхронного запроса, запущенного ранее вызовом `dblink_send_query`. Если запрос ещё выполняется, `dblink_get_result` будет ждать его завершения.

Аргументы

connname

Имя используемого подключения.

fail_on_error

Если равен true (это значение по умолчанию), в случае ошибки, выданной на удалённой стороне соединения, ошибка также выдаётся локально. Если равен false, удалённая ошибка выдаётся локально как ЗАМЕЧАНИЕ, и функция не возвращает строки.

Возвращаемое значение

Для асинхронного запроса (то есть, SQL-оператора, возвращающего строки) эта функция выдаёт строки, полученные в результате запроса. Чтобы использовать эту функцию, вы должны задать ожидаемый набор столбцов, как ранее говорилось в описании `dblink`.

Для асинхронной команды (то есть, SQL-оператора, не возвращающего строки), эта функция возвращает одну строку с одним текстовым столбцом, содержащим строку состояния команды. Для такого вызова в предложении `FROM` так же необходимо определить, что результат будет содержать один текстовый столбец.

Замечания

Эта функция *должна* вызываться, если `dblink_send_query` возвращает 1. Её нужно вызывать по одному разу для каждого отправленного запроса, а затем ещё раз для получения пустого набора данных, прежде чем подключением можно будет пользоваться снова.

Когда используются `dblink_send_query` и `dblink_get_result`, подсистема dblink получает весь набор удалённых результатов, прежде чем передавать его для локальной обработки. Если запрос возвращает большое количество строк, это может занимать много памяти в локальном сеансе. Поэтому может быть лучше открыть такой запрос как курсор, вызвав `dblink_open`, а затем выбирать результаты удобоваримыми порциями. Кроме того, можно воспользоваться простой функцией `dblink()`, которая не допускает заполнения памяти, выгружая большие наборы результатов на диск.

Примеры

```
contrib_regression=# SELECT dblink_connect('dtest1', 'dbname=contrib_regression');
 dblink_connect
----------------
 OK
(1 row)

contrib_regression=# SELECT * FROM
```

```
contrib_regression-# dblink_send_query('dtest1', 'select * from foo where f1 < 3') AS
 t1;
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
 f3 text[]);
 f1 | f2 |     f3
----+----+------------
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
 f3 text[]);
 f1 | f2 | f3
----+----+----
(0 rows)

contrib_regression=# SELECT * FROM
contrib_regression-# dblink_send_query('dtest1', 'select * from foo where f1 < 3;
 select * from foo where f1 > 6') AS t1;
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
 f3 text[]);
 f1 | f2 |     f3
----+----+------------
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
 f3 text[]);
 f1 | f2 |      f3
----+----+---------------
  7 | h  | {a7,b7,c7}
  8 | i  | {a8,b8,c8}
  9 | j  | {a9,b9,c9}
 10 | k  | {a10,b10,c10}
(4 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
 f3 text[]);
 f1 | f2 | f3
----+----+----
(0 rows)
```

dblink_cancel_query

dblink_cancel_query — отменяет любой активный запрос в заданном подключении

Синтаксис

```
dblink_cancel_query(text connname) returns text
```

Описание

Функция `dblink_cancel_query` пытается отменить любой запрос, выполняющийся через заданное подключение. Заметьте, что её вызов не обязательно будет успешным (например, потому что удалённый запрос уже завершился). Запрос отмены просто увеличивает шансы того, что выполняющийся запрос будет вскоре прерван. При этом всё равно нужно завершить обычную процедуру обработки запроса, например, вызвать `dblink_get_result`.

Аргументы

connname

Имя используемого подключения.

Возвращаемое значение

Возвращает `OK`, если запрос отмены был отправлен, либо текст сообщения об ошибке в случае неудачи.

Примеры

```
SELECT dblink_cancel_query('dtest1');
```

dblink_get_pkey

dblink_get_pkey — возвращает позиции и имена полей первичного ключа отношения

Синтаксис

```
dblink_get_pkey(text relname) returns setof dblink_pkey_results
```

Описание

Функция `dblink_get_pkey` выдаёт информацию о первичном ключе отношения в локальной базе данных. Иногда это полезно при формировании запросов, отправляемых в удалённые базы данных.

Аргументы

relname

Имя локального отношения, например `foo` или `myschema.mytab`. Заключите его в двойные кавычки, если это имя в смешанном регистре или содержит специальные символы, например `"FooBar"`; без кавычек эта строка приводится к нижнему регистру.

Возвращаемое значение

Возвращает одну строку для каждого поля первичного ключа, либо не возвращает строк, если в отношении нет первичного ключа. Тип результирующей строки определён как

```
CREATE TYPE dblink_pkey_results AS (position int, colname text);
```

В столбце `position` содержится число от 1 до *N*; это номер поля в первичном ключе, а не номер столбца в списке столбцов таблицы.

Примеры

```
CREATE TABLE foobar (
    f1 int,
    f2 int,
    f3 int,
    PRIMARY KEY (f1, f2, f3)
);
CREATE TABLE

SELECT * FROM dblink_get_pkey('foobar');
 position | colname
----------+---------
        1 | f1
        2 | f2
        3 | f3
(3 rows)
```

dblink_build_sql_insert

dblink_build_sql_insert — формирует оператор INSERT из локального кортежа, заменяя значения полей первичного ключа переданными альтернативными значениями

Синтаксис

```
dblink_build_sql_insert(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] src_pk_att_vals_array,
                        text[] tgt_pk_att_vals_array) returns text
```

Описание

Функция `dblink_build_sql_insert` может быть полезна при избирательной репликации локальной таблицы с удалённой базой данных. Она выбирает строку из локальной таблицы по заданному первичному ключу, а затем формирует SQL-команду `INSERT`, дублирующую эту строку, но заменяет в ней значения первичного ключа данными из последнего аргумента. (Чтобы получить точную копию строки, просто укажите одинаковые значения в двух последних аргументах.)

Аргументы

relname

Имя локального отношения, например `foo` или `myschema.mytab`. Заключите его в двойные кавычки, если это имя в смешанном регистре или содержит специальные символы, например `"FooBar"`; без кавычек эта строка приводится к нижнему регистру.

primary_key_attnums

Номера атрибутов (начиная с 1) полей первичного ключа, например `1 2`.

num_primary_key_atts

Число полей первичного ключа.

src_pk_att_vals_array

Значения полей первичного ключа, по которым будет выполняться поиск локального кортежа. Каждое поле здесь представляется в текстовом виде. Если локальной строки с этими значениями первичного ключа нет, выдаётся ошибка.

tgt_pk_att_vals_array

Значения полей первичного ключа, которые будут помещены в результирующую команду `INSERT`. Каждое поле представляется в текстовом виде.

Возвращаемое значение

Возвращает запрошенный SQL-оператор в текстовом виде.

Замечания

Начиная с PostgreSQL 9.0, номера атрибутов в *primary_key_attnums* воспринимаются как логические номера столбцов, соответствующие позициям столбцов в `SELECT * FROM relname`. Предыдущие версии воспринимали эти номера как физические позиции столбцов. Отличие этих подходов проявляется, когда на протяжении жизни таблицы из неё удаляются столбцы левее указанных.

Примеры

```
SELECT dblink_build_sql_insert('foo', '1 2', 2, '{"1", "a"}', '{"1", "b''a"}');
```

```
             dblink_build_sql_insert
------------------------------------------------
 INSERT INTO foo(f1,f2,f3) VALUES('1','b''a','1')
(1 row)
```

dblink_build_sql_delete

dblink_build_sql_delete — формирует оператор DELETE со значениями, передаваемыми для полей первичного ключа

Синтаксис

```
dblink_build_sql_delete(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] tgt_pk_att_vals_array) returns text
```

Описание

Функция `dblink_build_sql_delete` может быть полезна при избирательной репликации локальной таблицы с удалённой базой данных. Она формирует SQL-команду `DELETE`, которая удалит строку с заданными значениями первичного ключа.

Аргументы

relname

Имя локального отношения, например `foo` или `myschema.mytab`. Заключите его в двойные кавычки, если это имя в смешанном регистре или содержит специальные символы, например `"FooBar"`; без кавычек эта строка приводится к нижнему регистру.

primary_key_attnums

Номера атрибутов (начиная с 1) полей первичного ключа, например `1 2`.

num_primary_key_atts

Число полей первичного ключа.

tgt_pk_att_vals_array

Значения полей первичного ключа, которые будут использоваться в результирующей команде `DELETE`. Каждое поле представляется в текстовом виде.

Возвращаемое значение

Возвращает запрошенный SQL-оператор в текстовом виде.

Замечания

Начиная с PostgreSQL 9.0, номера атрибутов в *primary_key_attnums* воспринимаются как логические номера столбцов, соответствующие позициям столбцов в `SELECT * FROM relname`. Предыдущие версии воспринимали эти номера как физические позиции столбцов. Отличие этих подходов проявляется, когда на протяжении жизни таблицы из неё удаляются столбцы левее указанных.

Примеры

```
SELECT dblink_build_sql_delete('"MyFoo"', '1 2', 2, '{"1", "b"}');
           dblink_build_sql_delete
---------------------------------------------
 DELETE FROM "MyFoo" WHERE f1='1' AND f2='b'
(1 row)
```

dblink_build_sql_update

dblink_build_sql_update — формирует оператор UPDATE из локального кортежа, заменяя значения первичного ключа переданными альтернативными значениями

Синтаксис

```
dblink_build_sql_update(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] src_pk_att_vals_array,
                        text[] tgt_pk_att_vals_array) returns text
```

Описание

Функция `dblink_build_sql_update` может быть полезна при избирательной репликации локальной таблицы с удалённой базой данных. Она выбирает строку из локальной таблицы по заданному первичному ключу, а затем формирует SQL-команду `UPDATE`, дублирующую эту строку, но заменяющую в ней значения первичного ключа данными из последнего аргумента. (Чтобы получить точную копию строки, просто укажите одинаковые значения в двух последних аргументах.) Команда `UPDATE` всегда присваивает значения всем полям строки — основное отличие этой функции от `dblink_build_sql_insert` в том, что она предполагает, что целевая строка уже существует в удалённой таблице.

Аргументы

relname

Имя локального отношения, например `foo` или `myschema.mytab`. Заключите его в двойные кавычки, если это имя в смешанном регистре или содержит специальные символы, например `"FooBar"`; без кавычек эта строка приводится к нижнему регистру.

primary_key_attnums

Номера атрибутов (начиная с 1) полей первичного ключа, например `1 2`.

num_primary_key_atts

Число полей первичного ключа.

src_pk_att_vals_array

Значения полей первичного ключа, по которым будет выполняться поиск локального кортежа. Каждое поле здесь представляется в текстовом виде. Если локальной строки с этими значениями первичного ключа нет, выдаётся ошибка.

tgt_pk_att_vals_array

Значения полей первичного ключа, которые будут помещены в результирующую команду `UPDATE`. Каждое поле представляется в текстовом виде.

Возвращаемое значение

Возвращает запрошенный SQL-оператор в текстовом виде.

Замечания

Начиная с PostgreSQL 9.0, номера атрибутов в *primary_key_attnums* воспринимаются как логические номера столбцов, соответствующие позициям столбцов в `SELECT * FROM relname`. Предыдущие версии воспринимали эти номера как физические позиции столбцов. Отличие этих подходов проявляется, когда на протяжении жизни таблицы из неё удаляются столбцы левее указанных.

Примеры

```
SELECT dblink_build_sql_update('foo', '1 2', 2, '{"1", "a"}', '{"1", "b"}');
                   dblink_build_sql_update
-------------------------------------------------------------
 UPDATE foo SET f1='1',f2='b',f3='1' WHERE f1='1' AND f2='b'
(1 row)
```

F.13. dict_int

Модуль `dict_int` представляет собой пример дополнительного шаблона словаря для полнотекстового поиска. Этот словарь был создан для управляемой индексации целых чисел (со знаком и без); он позволяет индексировать такие числа и при этом избежать чрезмерного разрастания списка уникальных слов, что значительно увеличивает скорость поиска.

F.13.1. Конфигурирование

Этот словарь принимает два параметра:

- Параметр `maxlen` задаёт максимальное число цифр, из которого может состоять целое число. Значение по умолчанию — 6.
- Параметр `rejectlong` определяет, должны ли чрезмерно длинные числа усекаться или игнорироваться. Если `rejectlong` имеет значение `false` (по умолчанию), этот словарь возвращает первые `maxlen` цифр целого числа. Если `rejectlong` равен `true`, чрезмерное длинное целое воспринимается как стоп-слово, и в результате не индексируется. Заметьте, это означает, что такое целое нельзя будет найти.

F.13.2. Использование

При установке расширения `dict_int` в базе создаётся шаблон текстового поиска `intdict_template` и словарь `intdict` на его базе, с параметрами по умолчанию. Вы можете изменить параметры словаря, например так:

```
mydb# ALTER TEXT SEARCH DICTIONARY intdict (MAXLEN = 4, REJECTLONG = true);
ALTER TEXT SEARCH DICTIONARY
```

или создать новые словари на базе этого шаблона.

Протестировать этот словарь можно так:

```
mydb# select ts_lexize('intdict', '12345678');
 ts_lexize
-----------
 {123456}
```

Но для практического применения его нужно включить в конфигурацию текстового поиска, как описано в Главе 12. Это может выглядеть примерно так:

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR int, uint WITH intdict;
```

F.14. dict_xsyn

Модуль `dict_xsyn` (Extended Synonym Dictionary, расширенный словарь синонимов) представляет собой пример дополнительного шаблона словаря для полнотекстового поиска. Этот словарь заменяет слова группами их синонимов, что позволяет находить слово по одному из его синонимов.

F.14.1. Конфигурирование

Словарь `dict_xsyn` принимает следующие параметры:

- Параметр `matchorig` определяет, будет ли словарь принимать изначальное слово. По умолчанию он включён (имеет значение `true`).

- Параметр `matchsynonyms` определяет, будет ли словарь принимать синонимы. По умолчанию он отключён (имеет значение `false`).
- Параметр `keeporig` определяет, будет ли исходное слово включаться в вывод словаря. По умолчанию он включён (имеет значение `true`).
- Параметр `keepsynonyms` определяет, будут ли в вывод словаря включаться синонимы. По умолчанию он включён (имеет значение `true`).
- Параметр `rules` задаёт базовое имя файла со списком синонимов. Этот файл должен находиться в каталоге `$SHAREDIR/tsearch_data/` (где под `$SHAREDIR` понимается каталог с общими данными инсталляции Postgres Pro). Имя файла должно заканчиваться расширением `.rules` (которое не нужно указывать в параметре `rules`).

Файл правил имеет следующий формат:

- Каждая строка представляет группу синонимов для одного слова, которое задаётся первым в этой строке. Символы разделяются пробельными символами, так что строка выглядит так:

  ```
  word syn1 syn2 syn3
  ```

- Символ решётки (`#`) обозначает начало комментария. Он может находиться в любом месте строки. Следующая за ним часть строки игнорируется.

Пример словаря можно найти в файле `xsyn_sample.rules`, устанавливаемом в `$SHAREDIR/tsearch_data/`.

F.14.2. Использование

При установке расширения `dict_xsyn` в базе создаётся шаблон текстового поиска `xsyn_template` и словарь `xsyn` на его базе, с параметрами по умолчанию. Вы можете изменить параметры словаря, например так:

```
mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false);
ALTER TEXT SEARCH DICTIONARY
```

или создать новые словари на базе этого шаблона.

Протестировать этот словарь можно так:

```
mydb=# SELECT ts_lexize('xsyn', 'word');
      ts_lexize
-----------------------
 {syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'word');
      ts_lexize
-----------------------
 {word,syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false,
 MATCHSYNONYMS=true);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'syn1');
      ts_lexize
-----------------------
 {syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true,
 MATCHORIG=false, KEEPSYNONYMS=false);
ALTER TEXT SEARCH DICTIONARY
```

```
mydb=# SELECT ts_lexize('xsyn', 'syn1');
      ts_lexize
-----------------------
 {word}
```

Но для практического применения его нужно включить в конфигурацию текстового поиска, как описано в Главе 12. Это может выглядеть примерно так:

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR word, asciiword WITH xsyn, english_stem;
```

F.15. dump_stat

Модуль dump_stat предоставляет функции, позволяющие выгрузить и восстановить содержимое таблицы pg_statistic. При выполнении выгрузки/восстановления данных вы можете воспользоваться dump_stat для переноса существующей статистики на новый сервер, обойдясь без выполнения команды ANALYZE для всего кластера, что может значительно сократить время простоя для больших баз данных. Функция dump_statistic выдаёт операторы INSERT, которые затем можно применить к совместимой базе данных. Чтобы успешно восстановить статистические данные, вы должны установить это расширение и на исходном, и на целевом сервере, так как эти операторы будут задействовать функции, реализованные в dump_stat.

Заметьте, что определение таблицы pg_statistic может поменяться, в результате чего сгенерированный скрипт может оказаться несовместимым с будущими выпусками Postgres Pro.

F.15.1. Установка

Расширение dump_stat включено в состав Postgres Pro. Установив Postgres Pro, вы должны выполнить команду CREATE EXTENSION, чтобы подготовить dump_stat к работе, следующим образом:

```
CREATE EXTENSION dump_stat;
```

F.15.2. Функции

anyarray_to_text(array anyarray) returns text

Возвращает данный массив в виде значения text.

dump_statistic() returns setof text

Функция dump_statistic выгружает содержимое системного каталога pg_statistic. Она выдаёт INSERT для каждого кортежа в pg_statistic, кроме тех, что содержат статистические данные о таблицах в схемах information_schema и pg_catalog.

Оператор INSERT принимает следующую форму:

```
WITH upsert as (
  UPDATE pg_catalog.pg_statistic SET имя_столбца = выражение [, ...]
  WHERE starelid = t_relname::regclass
    AND to_attname(t_relname, staattnum) = t_attname
    AND to_atttype(t_relname, staattnum) = t_atttype
    AND stainherit = t_stainherit
  RETURNING *)
ins as (
  SELECT выражение [, ...]
  WHERE NOT EXISTS (SELECT * FROM upsert)
    AND to_attnum(t_relname, t_attname) IS NOT NULL
    AND to_atttype(t_relname, t_attname) = t_atttype)
INSERT INTO pg_catalog.pg_statistic SELECT * FROM ins;
```

где *выражение* может быть следующим:

```
array_in(текст_массива, имя_типа::regtype::oid, -1)
значение::имя_типа
```

Чтобы сохранить выданные операторы, направьте вывод psql в файл, используя стандартные возможности psql. Аргументы psql подробно рассматриваются в psql. Метакоманды, начинающиеся с обратной косой черты, не поддерживаются.

Например, чтобы сохранить статистику базы данных *имя_базы* в файле dump_stat.sql, выполните:

```
$ psql -XAtq -c "SELECT dump_statistic()" имя_базы > dump_stat.sql
```

`dump_statistic(schema_name text) returns setof text`

dump_statistic выгружает содержимое системного каталога pg_statistic. Она выдаёт оператор INSERT для каждого кортежа в pg_statistic, относящегося к какой-либо таблице в схеме schema_name.

`dump_statistic(schema_name text, table_name text) returns setof text`

dump_statistic выгружает содержимое системного каталога pg_statistic. Она формирует оператор INSERT для каждого кортежа в pg_statistic, относящегося к указанной таблице schema_name.table_name.

`dump_statistic(relation regclass) returns setof text`

dump_statistic выгружает содержимое системного каталога pg_statistic. Она формирует оператор INSERT для каждого кортежа в pg_statistic, содержащего статистические данные отношения relation.

`to_schema_qualified_operator(opid oid) returns text`

Выдаёт дополненное схемой имя оператора по его идентификатору opid. Например:

```
test=# SELECT to_schema_qualified_operator('+(int,int)'::regoperator);
          to_schema_qualified_operator
-------------------------------------------------
 pg_catalog.+(pg_catalog.int4, pg_catalog.int4)
(1 row)
```

`to_schema_qualified_type(typid oid) returns text`

Выдаёт дополненное схемой имя типа по его идентификатору typid.

`to_schema_qualified_relation(relid oid) returns text`

Выдаёт дополненное схемой имя отношения по его идентификатору relid.

`anyarray_elemtype(arr anyarray) returns oid`

Возвращает тип элемента массива, заданного по oid. Например:

```
test=# SELECT anyarray_elemtype(array_in('{1,2,3}', 'int'::regtype, -1));
 anyarray_elemtype
-------------------
                23
(1 row)
```

`to_attname(relation regclass, colnum int2) returns text`

Для отношения с заданным именем relation и номером столбца colnum возвращает имя столбца в виде значения text.

`to_attnum(relation regclass, col text) returns int2`

Для отношения с заданным именем relation и столбца с именем col возвращает номер столбца в виде значения int2.

`to_atttype(relation regclass, col text) returns text`

Для отношения с заданным именем `relation` и столбца с именем `col` возвращает дополненное схемой имя типа столбца в виде значения `text`.

`to_atttype(relation regclass, colnum int2) returns text`

Для отношения с заданным именем `relation` и столбца с номером `colnum` возвращает дополненное схемой имя типа столбца в виде значения `text`.

`to_namespace(nsp text) returns oid`

`to_namespace` повторяет поведение приведения к типу `regnamespace`, которое отсутствует в PostgreSQL версии 9.4 (и предыдущих). Эта функция возвращает `oid` данной схемы.

`get_namespace(relation oid) returns oid`

`get_namespace` возвращает схему заданного отношения в виде `oid`.

F.16. earthdistance

Модуль `earthdistance` реализует два разных варианта вычисления ортодромии (расстояния между точками на поверхности Земли). Первый описанный вариант зависит от модуля `cube`. Второй вариант основан на встроенном типе данных `point`, в котором в качестве координат задаются широта и долгота.

В этом модуле Земля считается идеальной сферой. (Если для вас это слишком грубо, обратите внимание на проект *PostGIS*.)

Прежде чем устанавливать `earthdistance`, вы должны установить модуль `cube` (хотя можно воспользоваться указанием `CASCADE` команды `CREATE EXTENSION` и установить сразу оба расширения).

> **Внимание**
>
> Расширения `earthdistance` и `cube` настоятельно рекомендуется устанавливать в одну схему, и при этом в данной схеме недоверенные пользователи не должны в настоящем и будущем иметь право CREATE. В противном случае, если в схеме `earthdistance` окажутся объекты, созданные злонамеренным пользователем, возможна угроза безопасности. Более того, используя функции `earthdistance` после установки расширения, следует ограничивать путь поиска только доверенными схемами.

F.16.1. Земные расстояния по кубам

Данные хранятся в кубах, представляющих точки (оба угла куба совпадают) по 3 координатам, выражающим смещения x, y и z от центра Земли. Этот модуль предоставляет домен `earth` на базе `cube`, включающий проверки того, что значение соответствует этим ограничениям и представляет точку, достаточно близкую к сферической поверхности Земли.

Радиус Земли выдаёт функция `earth()` (в метрах). Изменив одну эту функцию, вы можете сделать так, чтобы модуль работал с другими единицами, либо выдать другое значение радиуса, которое кажется вам более подходящим.

Этот пакет может также применяться и для астрономических расчётов. Астрономы обычно меняют функцию `earth()`, чтобы она возвращала радиус, равный `180/pi()`, и расстояния в результате выдавались в градусах.

В этом модуле реализованы функции для ввода данных, выражающих широту и долготу (в градусах), для вывода ширины и долготы, для вычисления ортодромии между двумя точками и простого указания окружающего прямоугольника, что полезно для поиска по индексу.

Предоставляемые этим модулем функции показаны в Таблица F.6.

Таблица F.6. Функции земных расстояний по кубам

Функция	Возвращает	Описание
`earth()`	`float8`	Возвращает предполагаемый радиус Земли.
`sec_to_gc(float8)`	`float8`	Переводит расстояние по обычной прямой (по секущей) между двумя точками на поверхности Земли в расстояние между ними по сфере.
`gc_to_sec(float8)`	`float8`	Переводит расстояние по сфере между двумя точками на поверхности Земли в расстояние по обычной прямой (по секущей) между ними.
`ll_to_earth(float8, float8)`	`earth`	Возвращает положение точки на поверхности Земли по заданной широте (аргумент 1) и долготе (аргумент 2) в градусах.
`latitude(earth)`	`float8`	Возвращает широту (в градусах) точки на поверхности Земли.
`longitude(earth)`	`float8`	Возвращает долготу (в градусах) точки на поверхности Земли.
`earth_distance(earth, earth)`	`float8`	Возвращает расстояние по сфере между двумя точками на поверхности Земли.
`earth_box(earth, float8)`	`cube`	Возвращает охватывающий куб, подходящий для поиска по индексу с применением оператора кубов `@>` точек в пределах заданной ортодромии от цели. Некоторые точки в этом кубе будут отстоять от цели дальше, чем на заданную ортодромию, поэтому в запрос нужно включить вторую проверку с функцией `earth_distance` .

F.16.2. Земные расстояния по точкам

Вторая часть этого модуля основана на представлении точек на Земле в виде значений типа `point`, в которых первый компонент представляет долготу в градусах, а второй — широту. Точки воспринимаются как (долгота, широта), а не наоборот, так как долгота ближе к интуитивному представлению как оси X, а широта — оси Y.

В модуле реализован один оператор, показанный в Таблице F.7.

Таблица F.7. Операторы земных расстояний по точкам

Оператор	Возвращает	Описание
`point <@> point`	`float8`	Выдаёт расстояние в сухопутных милях между точками на поверхности Земли.

Заметьте, что в этой части модуля, в отличие от части, построенной на `cube`, единицы зашиты жёстко: изменение функции `earth()` не повлияет на результат этого оператора.

Представление в виде долготы/широты плохо тем, что вам придётся учитывать граничные условия возле полюсов и в районе +/- 180 градусов долготы. Представление на базе `cube` лишено таких нарушений непрерывности.

F.17. fasttrun

Модуль `fasttrun` предоставляет транзакционно-небезопасную функцию для усечения временных таблиц, предотвращающую разрастание каталога pg_class.

Этот модуль требуется для поддержки системы 1С:Предприятие.

Операция быстрого усечения не является транзакционной, так что её действие нельзя отменить и оно немедленно становится видимым во всех сеансах независимо от уровня изоляции.

F.17.1. Функция

Пример вызова функции:

```
        select fasttruncate('TABLE_NAME');
```

F.17.2. Тестовый пример

Для тестирования вы можете использовать этот пример:

```
create or replace function f() returns void as $$
begin
  for i in 1..1000
    loop
      PERFORM fasttruncate('tt1');
    end loop;
  end;
$$ language plpgsql;
```

F.17.3. Авторы

Фёдор Сигаев `<teodor@sigaev.ru>`

F.18. file_fdw

Модуль `file_fdw` реализует обёртку сторонних данных `file_fdw`, с помощью которой можно обращаться к файлам данных в файловой системе сервера или выполнять программы на сервере и читать их вывод. Файлы и вывод программ должны быть в формате, который понимает команда `COPY FROM`; он рассматривается в описании COPY. В настоящее время файлы доступны только для чтения.

Для сторонней таблицы, создаваемой через эту обёртку, можно задать следующие параметры:

`filename`

Определяет имя файла, который нужно прочитать. При указании относительного пути он рассматривается от каталога данных. Вы должны определить либо параметр `filename`, либо `program`, но не оба сразу.

`program`

Определяет команду, которая будет выполнена. Поток стандартного вывода этой команды будет прочитан так же, как и с `COPY FROM PROGRAM`. Необходимо определить либо параметр `program`, либо `filename`, но не оба сразу.

format

Определяет формат файла (аналогично указанию FORMAT в команде COPY).

header

Указывает, что данные содержат строку заголовка с именами столбцов (аналогично указанию HEADER в команде COPY).

delimiter

Задаёт символ, разделяющий столбцы в данных (аналогично указанию DELIMITER в команде COPY).

quote

Задаёт символ, используемый для заключения данных в кавычки (аналогично указанию QUOTE в команде COPY).

escape

Задаёт символ, используемый для экранирования данных (аналогично указанию ESCAPE в команде COPY).

null

Определяет строку, задающую значение NULL в данных (аналогично указанию NULL в команде COPY).

encoding

Задаёт кодировку данных (аналогично указанию ENCODING в команде COPY).

Заметьте, что хотя COPY принимает указания, такие как HEADER, без соответствующего значения, синтаксис обёртки сторонних данных требует, чтобы значение присутствовало во всех случаях. Чтобы активировать указания COPY, которым значение обычно не передаётся, им можно просто передать значение TRUE, так как все они являются логическими.

Для столбцов сторонней таблицы, создаваемой через эту обёртку, можно задать следующие параметры:

force_not_null

Логическое значение. Если true, то значение столбца не должно сверяться со значением NULL (заданным в параметре null на уровне таблицы). Аналогично включению столбца в список указания FORCE_NOT_NULL команды COPY.

force_null

Логическое значение. Если true, значения столбцов нужно сверять со значением NULL (заданным в параметре NULL), даже если они заключены в кавычки. Без этого параметра только значения без кавычек, соответствующие значению null, будут возвращаться как NULL. Аналогично включению столбца в список указания FORCE_NULL команды COPY.

В настоящий момент file_fdw не поддерживает указания OIDS и FORCE_QUOTE команды COPY.

Перечисленные параметры применимы только для сторонних таблиц или их столбцов. Их нельзя указать для обёртки сторонних данных file_fdw, серверов или сопоставлений пользователей, использующих эту обёртку.

Для изменения параметров, определяемых для таблицы, требуются права суперпользователя. Это сделано в целях безопасности: только суперпользователь должен выбирать, какой файл читать или какую программу запускать. В принципе право изменения остальных параметров можно предоставить и не суперпользователям, но в настоящий момент это не реализовано.

Задавая параметр `program`, помните, что эта строка выполняется оболочкой ОС. Если вы хотите передавать заданной команде параметры из недоверенного источника, позаботьтесь об исключении или экранировании всех символов, которые могут иметь особое назначение в оболочке. По соображениям безопасности лучше, чтобы эта командная строка была фиксированной или как минимум в ней не передавались данные, поступающие от пользователя.

Для сторонних таблиц, работающих через `file_fdw`, команда `EXPLAIN` показывает имя используемого файла или запускаемой программы. Если не указывать `COSTS OFF`, то выводится и размер файла (в байтах).

Пример F.1. Создание сторонней таблицы для журнала сервера Postgres Pro

Одно из очевидных применений `file_fdw` — это предоставление доступа к журналу сообщений Postgres Pro как к таблице. Для этого необходимо предварительно настроить вывод сообщений в файл CSV (дальше мы будем считать, что это файл `pglog.csv`). Сначала установите расширение `file_fdw`:

```
CREATE EXTENSION file_fdw;
```

Затем создайте сторонний сервер:

```
CREATE SERVER pglog FOREIGN DATA WRAPPER file_fdw;
```

Всё готово для создания сторонней таблицы. В команде `CREATE FOREIGN TABLE` нужно перечислить столбцы таблицы, указать файл CSV и его формат:

```
CREATE FOREIGN TABLE pglog (
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text
) SERVER pglog
OPTIONS ( filename '/home/josh/data/log/pglog.csv', format 'csv' );
```

Вот и всё. Теперь для просмотра журнала сервера можно просто выполнять запросы к таблице. В производственной среде, разумеется, ещё потребуется как-то учесть ротацию файлов журнала.

F.19. fulleq

Модуль `fulleq` предоставляет дополнительный оператор равенства для совместимости с Microsoft SQL Server.

Этот модуль требуется для поддержки системы 1С:Предприятие.

F.19.1. Обзор

Оператор равенства в Postgres Pro определён так, что результатом сравнения двух NULL оказывается NULL. Однако в СУБД семейства Microsoft SQL исторически оператор равенства ведёт себя по-другому и возвращает TRUE, когда оба его операнда равны NULL. Данный модуль предоставляет такой оператор с поведением, принятым в MS SQL.

F.19.2. Оператор fulleq

Оператор == определён для следующих типов данных:

- bool
- bytea
- char
- name
- int2
- int4
- int8
- int2vector
- text
- oid
- xid
- cid
- oidvector
- float4
- float8
- abstime
- reltime
- macaddr
- inet
- cidr
- varchar
- date
- time
- timestamp
- timestamptz
- interval
- timetz

F.19.3. Авторы

Фёдор Сигаев <teodor@sigaev.ru>

F.20. fuzzystrmatch

Модуль fuzzystrmatch содержит несколько функций для вычисления схожести и расстояния между строками.

> **Внимание**
>
> В настоящее время функции soundex, metaphone, dmetaphone и dmetaphone_alt плохо работают с многобайтными кодировками (в частности, с UTF-8).

F.20.1. Soundex

Система Soundex позволяет вычислить похожие по звучанию имена, приводя их к одинаковым кодам. Изначально она использовалась для обработки данных переписи населения США в 1880, 1900 и 1910 г. Заметьте, что эта система не очень полезна для неанглоязычных имён.

Модуль fuzzystrmatch предоставляет две функции для работы с кодами Soundex:

```
soundex(text) returns text
difference(text, text) returns int
```

Функция soundex преобразует строку в код Soundex. Функция difference преобразует две строки в их коды Soundex и затем сообщает количество совпадающих позиций в этих кодах. Так как коды Soundex состоят из четырёх символов, результатом может быть число от нуля до четырёх (0 обозначает полное несоответствие, а 4 — точное совпадение). (Таким образом, имя этой функции не вполне корректное — лучшим именем для неё было бы similarity.)

Несколько примеров использования:

```
SELECT soundex('hello world!');

SELECT soundex('Anne'), soundex('Ann'), difference('Anne', 'Ann');
SELECT soundex('Anne'), soundex('Andrew'), difference('Anne', 'Andrew');
SELECT soundex('Anne'), soundex('Margaret'), difference('Anne', 'Margaret');

CREATE TABLE s (nm text);

INSERT INTO s VALUES ('john');
INSERT INTO s VALUES ('joan');
INSERT INTO s VALUES ('wobbly');
INSERT INTO s VALUES ('jack');

SELECT * FROM s WHERE soundex(nm) = soundex('john');

SELECT * FROM s WHERE difference(s.nm, 'john') > 2;
```

F.20.2. Левенштейн

Эта функция вычисляет расстояние Левенштейна между двумя строками:

```
levenshtein(text source, text target, int ins_cost, int del_cost, int sub_cost) returns
 int
levenshtein(text source, text target) returns int
levenshtein_less_equal(text source, text target, int ins_cost, int del_cost, int
 sub_cost, int max_d) returns int
levenshtein_less_equal(text source, text target, int max_d) returns int
```

И в source, и в target может быть передана любая строка, отличная от NULL, не длиннее 255 символов. Параметры стоимости (ins_cost, del_cost, sub_cost) определяют цену добавления,

удаления или замены символов, соответственно. Эти параметры можно опустить, как во второй версии функции; в этом случае все они по умолчанию равны 1.

Функция `levenshtein_less_equal` является ускоренной версией функции Левенштейна, предназначенной для использования, только когда интерес представляют небольшие расстояния. Если фактическое расстояние меньше или равно `max_d`, то `levenshtein_less_equal` возвращает точное его значение; в противном случае она возвращает значение, большее чем `max_d`. Если значение `max_d` отрицательное, она работает так же, как функция `levenshtein`.

Примеры:

```
test=# SELECT levenshtein('GUMBO', 'GAMBOL');
 levenshtein
-------------
           2
(1 row)

test=# SELECT levenshtein('GUMBO', 'GAMBOL', 2,1,1);
 levenshtein
-------------
           3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive',2);
 levenshtein_less_equal
------------------------
                      3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive',4);
 levenshtein_less_equal
------------------------
                      4
(1 row)
```

F.20.3. Metaphone

Metaphone, как и Soundex, построен на идее составления кода, представляющего входную строку. Две строки признаются похожими, если их коды совпадают.

Эта функция вычисляет код метафона входной строки:

```
metaphone(text source, int max_output_length) returns text
```

В качестве `source` должна передаваться строка, отличная от NULL, не длиннее 255 символов. Параметр `max_output_length` задаёт максимальную длину выходного кода метафона; если код оказывается длиннее, он обрезается до этой длины.

Пример:

```
test=# SELECT metaphone('GUMBO', 4);
 metaphone
-----------
 KM
(1 row)
```

F.20.4. Double Metaphone

Алгоритм Double Metaphone (Двойной метафон) вычисляет две строки «похожего звучания» для заданной строки — «первичную» и «альтернативную». В большинстве случаев они совпадают, но

для неанглоязычных имён в особенности они могут быть весьма различными, в зависимости от произношения. Эти функции вычисляют первичный и альтернативный коды:

```
dmetaphone(text source) returns text
dmetaphone_alt(text source) returns text
```

Длина входных строк может быть любой.

Пример:

```
test=# select dmetaphone('gumbo');
 dmetaphone
------------
 KMP
(1 row)
```

F.21. hstore

Этот модуль реализует тип данных `hstore` для хранения пар ключ-значение внутри одного значения Postgres Pro. Это может быть полезно в самых разных сценариях, например для хранения строк со множеством редко анализируемых атрибутов или частично структурированных данных. Ключи и значения задаются простыми текстовыми строками.

F.21.1. Внешнее представление `hstore`

Текстовое представление типа `hstore`, применяемое для ввода и вывода, включает ноль или более пар *`ключ`* `=>` *`значение`*, разделённых запятыми. Несколько примеров:

```
k => v
foo => bar, baz => whatever
"1-a" => "anything at all"
```

Порядок пар не имеет значения (и может не воспроизводиться при выводе). Пробелы между парами и вокруг знака `=>` игнорируются. Ключи и значения, содержащие пробелы, запятые и знаки `=` или `>`, нужно заключать в двойные кавычки. Если в ключ или значение нужно вставить символ кавычек или обратную косую черту, добавьте перед ним обратную косую черту.

Все ключи в `hstore` уникальны. Если вы объявите тип `hstore` с дублирующимися ключами, в `hstore` будет сохранён только один ключ без гарантии определённого выбора:

```
SELECT 'a=>1,a=>2'::hstore;
  hstore
----------
 "a"=>"1"
```

В качестве значения (но не ключа) может задаваться SQL `NULL`. Например:

```
key => NULL
```

В ключевом слове `NULL` регистр не имеет значения. Если требуется, чтобы текст `NULL` воспринимался как обычная строка «NULL», заключите его в кавычки.

> **Примечание**
>
> Учтите, что когда текстовый формат `hstore` используется для ввода данных, он применяется *до* обработки кавычек или спецсимволов. Таким образом, если значение `hstore` передаётся в параметре, дополнительная обработка не требуется. Но если вы передаёте его в виде строковой константы, то все символы апострофов и (в зависимости от параметра конфигурации `standard_conforming_strings`) обратной косой черты нужно корректно экранировать. Подробнее о записи строковых констант можно узнать в Подразделе 4.1.2.1.

При выводе значения и ключи всегда заключаются в кавычки, даже когда без этого можно обойтись.

F.21.2. Операторы и функции `hstore`

Реализованные в модуле `hstore` операторы перечислены в Таблице F.8, функции — в Таблице F.9.

Таблица F.8. Операторы `hstore`

Оператор	Описание	Пример	Результат
`hstore -> text`	выдаёт значение для ключа (или `NULL` при его отсутствии)	`'a=>x, b=>y'::hstore -> 'a'`	`x`
`hstore -> text[]`	выдаёт значения для ключей (или `NULL` при их отсутствии)	`'a=>x, b=>y, c=>z'::hstore -> ARRAY['c','a']`	`{"z","x"}`
`hstore || hstore`	объединяет два набора `hstore`	`'a=>b, c=>d'::hstore || 'c=>x, d=>q'::hstore`	`"a"=>"b", "c"=>"x", "d"=>"q"`
`hstore ? text`	набор `hstore` включает ключ?	`'a=>1'::hstore ? 'a'`	`t`
`hstore ?& text[]`	набор `hstore` включает все указанные ключи?	`'a=>1, b=>2'::hstore ?& ARRAY['a','b']`	`t`
`hstore ?| text[]`	набор `hstore` включает какой-либо из указанных ключей?	`'a=>1, b=>2'::hstore ?| ARRAY['b','c']`	`t`
`hstore @> hstore`	левый операнд включает правый?	`'a=>b, b=>1, c=>NULL'::hstore @> 'b=>1'`	`t`
`hstore <@ hstore`	левый операнд включён в правый?	`'a=>c'::hstore <@ 'a=>b, b=>1, c=>NULL'`	`f`
`hstore - text`	удаляет ключ из левого операнда	`'a=>1, b=>2, c=>3'::hstore - 'b'::text`	`"a"=>"1", "c"=>"3"`
`hstore - text[]`	удаляет ключи из левого операнда	`'a=>1, b=>2, c=>3'::hstore - ARRAY['a','b']`	`"c"=>"3"`
`hstore - hstore`	удаляет соответствующие пары из левого операнда	`'a=>1, b=>2, c=>3'::hstore - 'a=>4, b=>2'::hstore`	`"a"=>"1", "c"=>"3"`
`record #= hstore`	заменяет поля в `record` соответствующими значениями из `hstore`	см. раздел Примеры	
`%% hstore`	преобразует `hstore` в массив перемежающихся ключей и значений	`%% 'a=>foo, b=>bar'::hstore`	`{a,foo,b,bar}`
`%# hstore`	преобразует `hstore` в двумерный массив ключей-значений	`%# 'a=>foo, b=>bar'::hstore`	`{{a,foo},{b,bar}}`

Примечание

До версии PostgreSQL 8.2 операторы включения `@>` и `<@` обозначались соответственно как `@` и `~`. Эти имена по-прежнему действуют, но считаются устаревшими и в конце концов будут упразднены. Заметьте, что старые имена произошли из соглашения, которому раньше следовали геометрические типы данных!

Таблица F.9. Функции `hstore`

Функция	Тип результата	Описание	Пример	Результат
`hstore(record)`	`hstore`	формирует `hstore` из записи или кортежа	`hstore(ROW(1, 2))`	`f1=>1,f2=>2`
`hstore(text[])`	`hstore`	формирует `hstore` из массива, который может содержать попарно ключи-значения, либо быть двумерным массивом	`hstore( ARRAY['a','1', 'b','2']) || hstore( ARRAY[['c', '3'],['d', '4']])`	`a=>1, b=>2, c=>3, d=>4`
`hstore(text[], text[])`	`hstore`	формирует `hstore` из отдельных массивов ключей и значений	`hstore( ARRAY['a', 'b'], ARRAY['1', '2'])`	`"a"=>"1", "b"=>"2"`
`hstore(text, text)`	`hstore`	формирует `hstore` с одним элементом	`hstore('a', 'b')`	`"a"=>"b"`
`akeys(hstore)`	`text[]`	выдаёт ключи `hstore` в виде массива	`akeys('a=>1, b=>2')`	`{a,b}`
`skeys(hstore)`	`setof text`	выдаёт ключи `hstore` в виде множества	`skeys('a=>1, b=>2')`	`a` `b`
`avals(hstore)`	`text[]`	выдаёт значения `hstore` в виде массива	`avals('a=>1, b=>2')`	`{1,2}`
`svals(hstore)`	`setof text`	выдаёт значения `hstore` в виде множества	`svals('a=>1, b=>2')`	`1` `2`
`hstore_to_array(hstore)`	`text[]`	выдаёт ключи и значения `hstore` в виде массива перемежающихся ключей и значений	`hstore_to_array('a=>1, b=>2')`	`{a,1,b,2}`
`hstore_to_matrix(hstore)`	`text[]`	выдаёт ключи и значения `hstore` в виде двумерного массива	`hstore_to_matrix('a=>1, b=>2')`	`{{a,1},{b,2}}`

Функция	Тип результата	Описание	Пример	Результат
hstore_to_json(hstore)	json	выдаёт hstore в виде значения json, преобразуя все отличные от NULL значения в строки JSON	hstore_to_json('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')	{"a key": "1", "b": "t", "c": null, "d": "12345", "e": "012345", "f": "1.234", "g": "2.345e+4"}
hstore_to_jsonb(hstore)	jsonb	выдаёт hstore в виде значения jsonb, преобразуя все отличные от NULL значения в строки JSON	hstore_to_jsonb('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')	{"a key": "1", "b": "t", "c": null, "d": "12345", "e": "012345", "f": "1.234", "g": "2.345e+4"}
hstore_to_json_loose(hstore)	json	выдаёт hstore в виде значения json, по возможности распознавая числовые и логические значения и передавая их в JSON без кавычек	hstore_to_json_loose('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')	{"a key": 1, "b": true, "c": null, "d": 12345, "e": "012345", "f": 1.234, "g": 2.345e+4}
hstore_to_jsonb_loose(hstore)	jsonb	выдаёт hstore в виде значения jsonb, по возможности распознавая числовые и логические значения и передавая их в JSON без кавычек	hstore_to_jsonb_loose('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')	{"a key": 1, "b": true, "c": null, "d": 12345, "e": "012345", "f": 1.234, "g": 2.345e+4}
slice(hstore, text[])	hstore	извлекает подмножество из hstore	slice('a=>1, b=>2, c=>3'::hstore, ARRAY['b','c', 'x'])	"b"=>"2", "c"=>"3"
each(hstore)	setof(key text, value text)	выдаёт ключи и значения hstore в виде множества	select * from each('a=>1, b=>2')	key | value -----+------- a | 1 b | 2
exist(hstore, text)	boolean	набор hstore включает ключ?	exist('a=>1', 'a')	t
defined(hstore, text)	boolean	набор hstore включает для ключа значение, отличное от NULL?	defined('a=>NULL','a')	f

Функция	Тип результата	Описание	Пример	Результат
delete(hstore, text)	hstore	удаляет пару с соответствующим ключом	delete('a=>1, b=>2','b')	"a"=>"1"
delete(hstore, text[])	hstore	удаляет пары с соответствующими ключами	delete('a=>1, b=>2,c=>3', ARRAY['a', 'b'])	"c"=>"3"
delete(hstore, hstore)	hstore	удаляет пары, соответствующие парам во втором аргументе	delete('a=>1, b=>2','a=>4, b=>2'::hstore)	"a"=>"1"
populate_ record(record, hstore)	record	заменяет поля в record соответствующими значениями из hstore	см. раздел Примеры	

Примечание

Функция hstore_to_json применяется, когда значение hstore нужно привести к json. Подобным образом, hstore_to_jsonb применяется, когда значение hstore нужно привести к jsonb.

Примечание

Функция populate_record на самом деле объявлена как принимающая в первом аргументе anyelement, а не record, но если ей будет передан не тип записи, она выдаст ошибку.

F.21.3. Индексы

Тип hstore поддерживает индексы GiST и GIN для операторов @>, ?, ?& и ?|. Например:

```
CREATE INDEX hidx ON testhstore USING GIST (h);

CREATE INDEX hidx ON testhstore USING GIN (h);
```

Тип hstore также поддерживает индексы btree и hash для оператора =. Это позволяет объявлять столбцы hstore как уникальные (UNIQUE) и использовать их в выражениях GROUP BY, ORDER BY или DISTINCT. Порядок сортировки значений hstore не имеет практического смысла, но эти индексы могут быть полезны для поиска по равенству. Индексы для сравнений (с помощью =) можно создать так:

```
CREATE INDEX hidx ON testhstore USING BTREE (h);

CREATE INDEX hidx ON testhstore USING HASH (h);
```

F.21.4. Примеры

Добавление ключа или изменение значения для существующего ключа:

```
UPDATE tab SET h = h || hstore('c', '3');
```

Удаление ключа:

```
UPDATE tab SET h = delete(h, 'k1');
```

Приведение типа `record` к типу `hstore`:

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT hstore(t) FROM test AS t;
                  hstore
-------------------------------------------
 "col1"=>"123", "col2"=>"foo", "col3"=>"bar"
(1 row)
```

Приведение типа `hstore` к предопределённому типу `record`:

```
CREATE TABLE test (col1 integer, col2 text, col3 text);

SELECT * FROM populate_record(null::test,
                              '"col1"=>"456", "col2"=>"zzz"');
 col1 | col2 | col3
------+------+------
  456 | zzz  |
(1 row)
```

Изменение существующей записи по данным из `hstore`:

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT (r).* FROM (SELECT t #= '"col3"=>"baz"' AS r FROM test t) s;
 col1 | col2 | col3
------+------+------
  123 | foo  | baz
(1 row)
```

F.21.5. Статистика

Тип `hstore`, вследствие присущей ему либеральности, может содержать множество самых разных ключей. Контроль допустимости ключей является задачей приложения. Следующие примеры демонстрируют несколько приёмов проверки ключей и получения статистики.

Простой пример:

```
SELECT * FROM each('aaa=>bq, b=>NULL, ""=>1');
```

С таблицей:

```
SELECT (each(h)).key, (each(h)).value INTO stat FROM testhstore;
```

Актуальная статистика:

```
SELECT key, count(*) FROM
  (SELECT (each(h)).key FROM testhstore) AS stat
  GROUP BY key
  ORDER BY count DESC, key;
    key    | count
-----------+-------
 line      |   883
 query     |   207
 pos       |   203
 node      |   202
```

```
 space     |   197
 status    |   195
 public    |   194
 title     |   190
 org       |   189
.................
```

F.21.6. Совместимость

Начиная с PostgreSQL 9.0, `hstore` использует внутреннее представление, отличающееся от предыдущих версий. Это не проблема при обновлении путём выгрузки/перезагрузки данных, так как текстовое представление (используемое при выгрузке) не меняется.

В случае двоичного обновления обратная совместимость поддерживается благодаря тому, что новый код понимает данные в старом формате. При таком обновлении возможно небольшое снижение производительности при обработке данных, которые ещё не были изменены новым кодом. Все значения в столбце таблицы можно обновить принудительно, выполнив следующий оператор `UPDATE`:

```
UPDATE tablename SET hstorecol = hstorecol || '';
```

Это можно сделать и так:

```
ALTER TABLE tablename ALTER hstorecol TYPE hstore USING hstorecol || '';
```

Вариант с командой `ALTER TABLE` требует блокировки таблицы в режиме `ACCESS EXCLUSIVE`, но не приводит к замусориванию таблицы старыми версиями строк.

F.21.7. Трансформации

Также имеются дополнительные расширения, реализующие трансформации типа `hstore` для языков PL/Perl и PL/Python. Расширения для PL/Perl называются `hstore_plperl` и `hstore_plperlu` для доверенного и недоверенного PL/Perl, соответственно. Если вы установите эти трансформации и укажете их при создании функции, значения `hstore` будут отображаться в хеши Perl. Расширения для PL/Python называются `hstore_plpythonu`, `hstore_plpython2u` и `hstore_plpython3u` (соглашения об именовании, принятые для интерфейса PL/Python, описаны в Разделе 43.1). Если вы воспользуетесь ими, значения `hstore` будут отображаться в словари Python.

> **Внимание**
>
> Расширения, реализующие трансформации, настоятельно рекомендуется устанавливать в одну схему с `hstore`. Выбор какой-либо другой схемы, которая может содержать объекты, созданные злонамеренным пользователем, чреват угрозами безопасности во время установки расширения.

F.21.8. Авторы

Олег Бартунов `<oleg@sai.msu.su>`, Москва, Московский Государственный Университет, Россия

Фёдор Сигаев `<teodor@sigaev.ru>`, Москва, ООО «Дельта-Софт», Россия

Дополнительные улучшения внёс Эндрю Гирт `<andrew@tao11.riddles.org.uk>`, Великобритания

F.22. Модули словарей Hunspell

Эти модули предоставляют словари Hunspell для разных языков. После установки модуля в базу командой `CREATE EXTENSION` в схеме public появляются объекты конфигурации и словаря текстового поиска.

Таблица F.10. Модули

Язык	Имя расширения	Имя словаря	Имя конфигурации
Американский английский	hunspell_en_us	english_hunspell	english_hunspell
Нидерландский	hunspell_nl_nl	dutch_hunspell	dutch_hunspell
Французский	hunspell_fr	french_hunspell	french_hunspell
Русский	hunspell_ru_ru	russian_hunspell	russian_hunspell

F.22.1. Примеры

Объекты текстового поиска будут созданы после установки модуля словаря. Мы можем проверить созданную конфигурацию:

```
SELECT * FROM ts_debug('english_hunspell', 'abilities');
   alias   |   description   |   token   |          dictionaries           |
 dictionary    |  lexemes
-----------+-----------------+-----------+---------------------------------
+-----------------+-----------
 asciiword | Word, all ASCII | abilities | {english_hunspell,english_stem} |
 english_hunspell | {ability}
(1 row)
```

Либо вы можете создать собственную конфигурацию текстового поиска. Например, с созданными словарями и словарём Snowball вы можете получить смешанную русско-английскую конфигурацию:

```
CREATE TEXT SEARCH CONFIGURATION russian_en (
  COPY = simple
);

ALTER TEXT SEARCH CONFIGURATION russian_en
  ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
  WITH english_hunspell, english_stem;

ALTER TEXT SEARCH CONFIGURATION russian_en
  ALTER MAPPING FOR word, hword, hword_part
  WITH russian_hunspell, russian_stem;
```

Создавать смешанные словари можно только для языков с различными алфавитами. Если у двух языков похожие алфавиты, Postgres Pro не сможет определить, какой словарь нужно использовать.

Конфигурация текстового поиска, созданная модулем словаря, непосредственно готова к использованию. Например, так можно искать определённые слова в этом тексте:

```
SELECT to_tsvector('english_hunspell', 'The blue whale is the largest animal');
               to_tsvector
-----------------------------------------
 'animal':7 'blue':2 'large':6 'whale':3
(1 row)
```

Поисковый запрос может выглядеть так:

```
SELECT to_tsvector('english_hunspell', 'The blue whale is the largest animal')
  @@ to_tsquery('english_hunspell', 'large & whale');
 ?column?
----------
 t
```

```
(1 row)
```

С такими конфигурациями можно искать текст, применяя индексы GIN или GIST. Например, если существует таблица с индексом GIN:

```
CREATE TABLE table1 (t varchar);
INSERT INTO table1 VALUES ('The blue whale is the largest animal');
CREATE INDEX t_idx ON table1 USING GIN (to_tsvector('english_hunspell', "t"));
```

Для неё можно выполнить следующий запрос:

```
SELECT * FROM table1 where to_tsvector('english_hunspell', t)
  @@ to_tsquery('english_hunspell', 'blue & animal');
                  t
-------------------------------------
 The blue whale is the largest animal
(1 row)
```

F.23. intagg

Модуль `intagg` предоставляет агрегатор и нумератор целых чисел. На данный момент имеются встроенные функции, предлагающие более широкие возможности, поэтому `intagg` считается устаревшим. Однако этот модуль продолжает существовать для обратной совместимости, теперь как набор обёрток встроенных функций.

F.23.1. Функции

Агрегатор реализуется функцией `int_array_aggregate(integer)`, которая выдаёт массив целых чисел, содержащий в точности те числа, что переданы ей. Это обёртка встроенной функции `array_agg`, которая делает то же самое для массива любого типа.

Нумератор реализуется функцией `int_array_enum(integer[])`, которая возвращает набор целых (`setof integer`). По сути его действие обратно действие агрегатора: получив массив целых, он разворачивает его в набор строк. Это оболочка функции `unnest`, которая делает то же самое для массива любого типа.

F.23.2. Примеры использования

Во многих СУБД есть понятие таблицы соотношений «один ко многим». Такая таблица обычно находится между двумя индексированными таблицами, например:

```
CREATE TABLE left (id INT PRIMARY KEY, ...);
CREATE TABLE right (id INT PRIMARY KEY, ...);
CREATE TABLE one_to_many(left INT REFERENCES left, right INT REFERENCES right);
```

Как правило, она используется так:

```
SELECT right.* from right JOIN one_to_many ON (right.id = one_to_many.right)
  WHERE one_to_many.left = item;
```

Этот запрос вернёт все элементы из таблицы справа для записи в таблице слева. Это очень распространённая конструкция в SQL.

Однако этот подход может вызывать затруднения с очень большим количеством записей в таблице `one_to_many`. Часто такое соединение влечёт сканирование индекса и выборку каждой записи в таблице справа для конкретного элемента слева. Если у вас динамическая система, с этим ничего не поделать. Но если какое-то множество данных довольно статическое, вы можете создать сводную таблицу, применив агрегатор.

```
CREATE TABLE summary AS
  SELECT left, int_array_aggregate(right) AS right
```

```
  FROM one_to_many
  GROUP BY left;
```

Эта команда создаст таблицу, содержащую одну строку для каждого элемента слева с массивом элементов справа. Она малополезна, пока не найден подходящий способ использования этого массива; именно для этого и нужен нумератор массива. Вы можете выполнить:

```
SELECT left, int_array_enum(right) FROM summary WHERE left = элемент;
```

Приведённый выше запрос с вызовом `int_array_enum` выдаёт те же результаты, что и

```
SELECT left, right FROM one_to_many WHERE left = элемент;
```

Отличие состоит в том, что запрос к сводной таблице должен выдать только одну строку таблицы, тогда как непосредственный запрос к `one_to_many` потребует сканирования индекса и выборки строки для каждой записи.

На тестовом компьютере команда `EXPLAIN` показала, что стоимость запроса снизилась с 8488 до 329. Исходный запрос выполнял соединение с таблицей `one_to_many` и был заменён на:

```
SELECT right, count(right) FROM
  ( SELECT left, int_array_enum(right) AS right
    FROM summary JOIN (SELECT left FROM left_table WHERE left = элемент) AS lefts
         ON (summary.left = lefts.left)
  ) AS list
  GROUP BY right
  ORDER BY count DESC;
```

F.24. intarray

Модуль `intarray` предоставляет ряд полезных функций и операторов для работы с массивами целых чисел без NULL. Также он поддерживает поиск по индексу для некоторых из этих операторов.

Все эти операции выдают ошибку, если в передаваемом массиве оказываются значения NULL.

Многие из этих операций имеют смысл только с одномерными массивами. Хотя им можно передать входной массив и большей размерности, значения будут считываться из него как из линейного массива в порядке хранения.

F.24.1. Функции и операторы `intarray`

Реализованные в модуле `intarray` функции перечислены в Таблице F.11, а операторы — в Таблице F.12.

Таблица F.11. Функции `intarray`

Функция	Тип результата	Описание	Пример	Результат
`icount(int[])`	`int`	число элементов в массиве	`icount('{1,2,3}'::int[])`	`3`
`sort(int[], text dir)`	`int[]`	сортирует массив — в *dir* должно задаваться `asc` (по возрастанию) или `desc` (по убыванию)	`sort('{1,2,3}'::int[], 'desc')`	`{3,2,1}`
`sort(int[])`	`int[]`	сортирует в порядке возрастания	`sort(array[11,77,44])`	`{11,44,77}`

Функция	Тип результата	Описание	Пример	Результат
`sort_asc(int[])`	`int[]`	сортирует в порядке возрастания		
`sort_desc(int[])`	`int[]`	сортирует в порядке убывания		
`uniq(int[])`	`int[]`	удаляет дубликаты	`uniq(sort('{1,2,3,2,1}'::int[]))`	`{1,2,3}`
`idx(int[], int item)`	`int`	индекс первого элемента, равного *item* (0, если такого нет)	`idx(array[11,22,33,22,11], 22)`	`2`
`subarray(int[], int start, int len)`	`int[]`	часть массива, начинающаяся с позиции *start* и состоящая из *len* элементов	`subarray('{1,2,3,2,1}'::int[], 2, 3)`	`{2,3,2}`
`subarray(int[], int start)`	`int[]`	часть массива, начинающаяся с позиции *start*	`subarray('{1,2,3,2,1}'::int[], 2)`	`{2,3,2,1}`
`intset(int)`	`int[]`	создаёт массив с одним элементом	`intset(42)`	`{42}`

Таблица F.12. Операторы `intarray`

Оператор	Возвращает	Описание
`int[] && int[]`	`boolean`	пересекается с — `true`, если массивы имеют минимум один общий элемент
`int[] @> int[]`	`boolean`	включает — `true`, если левый массив содержит правый массив
`int[] <@ int[]`	`boolean`	включается в — `true`, если левый массив содержится в правом массиве
`# int[]`	`int`	число элементов в массиве
`int[] # int`	`int`	индекс элемента (делает то же, что и функция `idx`)
`int[] + int`	`int[]`	вставляет элемент в массив (добавляет его в конец массива)
`int[] + int[]`	`int[]`	соединяет массивы (правый массив добавляется в конец левого)
`int[] - int`	`int[]`	удаляет из массива записи, равные правому аргументу
`int[] - int[]`	`int[]`	удаляет из левого массива элементы правого массива
`int[] | int`	`int[]`	объединение аргументов
`int[] | int[]`	`int[]`	объединение массивов

Оператор	Возвращает	Описание
`int[] & int[]`	`int[]`	пересечение массивов
`int[] @@ query_int`	`boolean`	`true`, если массив удовлетворяет запросу (см. ниже)
`query_int ~~ int[]`	`boolean`	`true`, если запросу удовлетворяет массив (коммутирующий оператор к `@@`)

(До версии PostgreSQL 8.2 операторы включения `@>` и `<@` обозначались соответственно как `@` и `~`. Эти имена по-прежнему действуют, но считаются устаревшими и в конце концов будут упразднены. Заметьте, что старые имена произошли из соглашения, которому раньше следовали ключевые геометрические типы данных!)

Операторы `&&`, `@>` и `<@` равнозначны встроенным операторам Postgres Pro с теми же именами, за исключением того, что они работают только с целочисленными массивами, не содержащими NULL, тогда как встроенные операторы работают с массивами любых типов. Благодаря этому ограничению, в большинстве случаев они работают быстрее, чем встроенные операторы.

Операторы `@@` и `~~` проверяют, удовлетворяет ли массив *запросу*, представляемому в виде значения специализированного типа данных `query_int`. *Запрос* содержит целочисленные значения, сравниваемые с элементами массива, возможно с использованием операторов `&` (AND), `|` (OR) и `!` (NOT). При необходимости могут использоваться скобки. Например, запросу `1&(2|3)` удовлетворяют запросы, которые содержат 1 и также содержат 2 или 3.

F.24.2. Поддержка индексов

Модуль `intarray` поддерживает индексы для операторов `&&`, `@>`, `<@` и `@@`, а также обычную проверку равенства массивов.

Модуль предоставляет два класса операторов GiST: `gist__int_ops` (используется по умолчанию), подходящий для маленьких и средних по размеру наборов данных, и `gist__intbig_ops`, применяющий сигнатуру большего размера и подходящий для индексации больших наборов данных (то есть столбцов, содержащих много различных значений массива). В этой реализации используется структура данных RD-дерева со встроенным сжатием с потерями.

Есть также нестандартный класс операторов GIN, `gin__int_ops`, поддерживающий те же операторы.

Выбор между индексами GiST и GIN зависит от относительных характеристик производительности GiST и GIN, которые здесь не рассматриваются.

F.24.3. Пример

```
-- сообщение может относиться к одной или нескольким «секциям»
CREATE TABLE message (mid INT PRIMARY KEY, sections INT[], ...);

-- создать специализированный индекс
CREATE INDEX message_rdtree_idx ON message USING GIST (sections gist__int_ops);

-- вывести сообщения из секций 1 или 2 — оператор пересечения
SELECT message.mid FROM message WHERE message.sections && '{1,2}';

-- вывести сообщения из секций 1 и 2 — оператор включения
SELECT message.mid FROM message WHERE message.sections @> '{1,2}';
```

```
-- тот же результат, но с оператором запроса
SELECT message.mid FROM message WHERE message.sections @@ '1&2'::query_int;
```

F.24.4. Авторы

Разработку осуществили Фёдор Сигаев (<teodor@sigaev.ru>) и Олег Бартунов (<oleg@sai.msu.su>). Дополнительные сведения можно найти на странице *http://www.sai.msu.su/~megera/postgres/gist/*. Андрей Октябрьский проделал отличную работу, добавив новые функции и операторы.

F.25. isn

Модуль `isn` предоставляет типы данных для следующих международных стандартов нумерации товаров: EAN13, UPC, ISBN (книги), ISMN (музыка) и ISSN (серийные номера). Номера проверяются на входе согласно жёстко заданному списку префиксов: этот список префиксов также применяется для группирования цифр при выводе. Так как время от времени появляются новые префиксы, этот список префиксов может устаревать. Ожидается, что будущая версия этого модуля будет получать список префиксов из одной или нескольких таблиц, благодаря чему его при необходимости смогут легко обновлять пользователи; однако в настоящее время этот список можно изменить, только модифицировав исходный код и перекомпилировав его. Также есть вероятность, что в будущем этот модуль лишится функций проверки префиксов и группирования цифр.

F.25.1. Типы данных

В Таблице F.13 перечислены типы данных, реализованные модулем `isn`.

Таблица F.13. Типы данных `isn`

Тип данных	Описание
`EAN13`	European Article Number (Европейский номер товара), всегда выводится в формате EAN13
`ISBN13`	International Standard Book Number (Международный стандартный книжный номер), выводимый в новом формате EAN13
`ISMN13`	International Standard Music Number (Международный стандартный музыкальный номер), выводимый в новом формате EAN13
`ISSN13`	International Standard Serial Number (Международный стандартный серийный номер), выводимый в новом формате EAN13
`ISBN`	Международный стандартный книжный номер, выводимый в старом коротком формате
`ISMN`	Международный стандартный музыкальный номер, выводимый в старом коротком формате
`ISSN`	Международный стандартный серийный номер, выводимый в старом коротком формате
`UPC`	Universal Product Code (Универсальный код товара)

Замечания:

1. Номера ISBN13, ISMN13, ISSN13 являются номерами EAN13.

2. Не все номера EAN13 (только их подмножество) представляют ISBN13, ISMN13 или ISSN13.

3. Некоторые номера ISBN13 можно вывести в формате ISBN.
4. Некоторые номера ISMN13 можно вывести в формате ISMN.
5. Некоторые номера ISSN13 можно вывести в формате ISSN.
6. Номера UPC являются подмножеством номеров EAN13 (по сути они являются номерами EAN13 без первой цифры `0`).
7. Любые номера UPC, ISBN, ISMN и ISSN можно представить как номера EAN13.

Внутри все эти типы представляются одинаково (64-битными целыми числами) и все они взаимозаменяемы. Различные типы введены для управления форматированием при выводе и для строгой проверки правильности ввода, что и определяет конкретный тип номера.

Типы `ISBN`, `ISMN` и `ISSN` выводят короткую версию числа (ISxN 10), когда это возможно, либо выбирают формат ISxN 13 для чисел, не умещающихся в короткую версию. Типы `EAN13`, `ISBN13`, `ISMN13` и `ISSN13` всегда выводят длинную версию ISxN (EAN13).

F.25.2. Приведения

Модуль `isn` предоставляет следующие пары приведений типов:

- ISBN13 <=> EAN13
- ISMN13 <=> EAN13
- ISSN13 <=> EAN13
- ISBN <=> EAN13
- ISMN <=> EAN13
- ISSN <=> EAN13
- UPC <=> EAN13
- ISBN <=> ISBN13
- ISMN <=> ISMN13
- ISSN <=> ISSN13

При приведении `EAN13` к другому типу выполняется проверка времени выполнения, соответствует ли исходное значение целевому типу, и если это не так, выдаётся ошибка. При других приведениях значения просто перемечаются, так что они успешно выполняются всегда.

F.25.3. Функции и операторы

Модуль `isn` предоставляет стандартные операторы сравнения плюс поддержку индексов по хешу и B-деревьев для этих типов данных. Кроме того, он реализует ряд специализированных функций; они перечислены в Таблице F.14. В этой таблице под `isn` понимается один из типов данных модуля.

Таблица F.14. Функции `isn`

Функция	Возвращает	Описание
`isn_weak(boolean)`	`boolean`	Устанавливает ослабленный режим проверки ввода (возвращает новое состояние)
`isn_weak()`	`boolean`	Выдаёт текущее состояние ослабленного режима
`make_valid(isn)`	`isn`	Делает неверный номер верным (сбрасывает флаг ошибки)

Функция	Возвращает	Описание
is_valid(isn)	boolean	Проверяет присутствие флага ошибки

Ослабленный режим позволяет вставлять в таблицу неверные значения. Под неверным значением понимается номер, в котором не сходится проверочная цифра, а не номер отсутствует вовсе.

Зачем может понадобиться ослабленный режим? Ну например, у вас может быть большой набор номеров ISBN, и очень многие из них по каким-то странным причинам содержат неверную контрольную цифру (возможно, номера были отсканированы с печатного листа и были распознаны неправильно, или они вводились вручную, кто знает...). В любом случае суть в том, что вы хотите с этим разобраться, но вам нужно загрузить все эти номера в базу данных, а затем, возможно, использовать дополнительное средство поиска неверных номеров в базе данных, чтобы проверить и исправить данные; так что, например, вам нужно будет выбрать все неверные номера в таблице.

Если попытаться вставить в таблицу неверный номер в ослабленном режиме проверки, номер будет вставлен с исправленной проверочной цифрой, но выводиться будет с восклицательным знаком (`!`) в конце, например `0-11-000322-5!`. Этот маркер ошибки можно проверить с помощью функции `is_valid` и очистить функцией `make_valid`.

Вы также можете принудительно вставить числа даже не в ослабленном режиме, добавив символ `!` в конце номера.

Есть ещё одна особенность — во вводимом номере вместо проверочной цифры можно указать `?` и нужная проверочная цифра будет вставлена автоматически.

F.25.4. Примеры

```
-- Использование типов напрямую:
SELECT isbn('978-0-393-04002-9');
SELECT isbn13('0901690546');
SELECT issn('1436-4522');

-- Приведение типов:
--  заметьте, что номер ean13 можно привести к другому типу, только когда
--  этот номер будет допустимым для целевого типа;
--  таким образом, это НЕ будет работать: select isbn(ean13('0220356483481'));
--  а это будет:
SELECT upc(ean13('0220356483481'));
SELECT ean13(upc('220356483481'));

-- Создание таблицы с одним столбцом, который будет содержать номера ISBN:
CREATE TABLE test (id isbn);
INSERT INTO test VALUES('9780393040029');

-- Автоматическое вычисление проверочных цифр (обратите внимание на '?'):
INSERT INTO test VALUES('220500896?');
INSERT INTO test VALUES('978055215372?');

SELECT issn('3251231?');
SELECT ismn('979047213542?');

-- Использование ослабленного режима:
SELECT isn_weak(true);
INSERT INTO test VALUES('978-0-11-000533-4');
INSERT INTO test VALUES('9780141219307');
INSERT INTO test VALUES('2-205-00876-X');
SELECT isn_weak(false);
```

```
SELECT id FROM test WHERE NOT is_valid(id);
UPDATE test SET id = make_valid(id) WHERE id = '2-205-00876-X!';

SELECT * FROM test;

SELECT isbn13(id) FROM test;
```

F.25.5. Библиография

Информация, на основе которой реализован этот модуль, была собрана с нескольких сайтов, включая:

- *https://www.isbn-international.org/*
- *https://www.issn.org/*
- *https://www.ismn-international.org/*
- *https://www.wikipedia.org/*

Префиксы, используемые для группирования цифр, были также взяты с:

- *http://www.gs1.org/productssolutions/idkeys/support/prefix_list.html*
- *http://en.wikipedia.org/wiki/List_of_ISBN_identifier_groups*
- *https://www.isbn-international.org/content/isbn-users-manual*
- *http://en.wikipedia.org/wiki/International_Standard_Music_Number*
- *http://www.ismn-international.org/ranges.html*

Алгоритмы реализованы со всей тщательностью и скрупулёзно сверены с алгоритмами, предлагаемыми в официальных руководствах ISBN, ISMN, ISSN.

F.25.6. Автор

Герман Мендез Браво (Kronuz), 2004 — 2006

Этот модуль написан под влиянием кода `isbn_issn` Гаретта А. Уоллмена.

F.26. jsquery

JsQuery — это язык запросов к данным `jsonb`. Его основное предназначение — предоставить дополнительную функциональность для `jsonb`, например, простой и эффективный способ поиска во вложенных объектах и массивах, а также дополнительные операторы сравнения с поддержкой индексов.

JsQuery реализован посредством типа данных `jsquery` (подобно `tsquery`) и оператора `@@` для применения запросов к `jsonb`.

F.26.1. Установка

Программный комплект Postgres Pro включает jsquery в качестве дополнительного модуля (contrib). Установив необходимые компоненты Postgres Pro Standard, создайте расширение jsquery следующим образом:

```
CREATE EXTENSION jsquery;
```

F.26.2. Язык запросов JSON

Расширение JsQuery содержит тип `jsquery`, представляющий весь запрос JSON в виде одного значения (как и `tsquery` для полнотекстового поиска). В этом запросе задаётся выражение, обращающееся к документам JSON.

Простое выражение записывается в виде *путь бинарный_оператор значение* или *путь унарный_оператор*. См. следующие примеры.

- `x = "abc"` — значение ключа "x" равно "abc";
- `$ @> [4, 5, "zzz"]` — документ JSON представляет собой массив, содержащий значения 4, 5 и "zzz";
- `"abc xyz" >= 10` — значение ключа "abc xyz" больше или равно 10;
- `volume IS NUMERIC` — ключ «volume» имеет числовой тип.
- `$ = true` — весь документ JSON представляет собой просто значение true.
- `similar_ids.@# > 5` - `similar_ids` является массивом или объектом с числом элементов больше 5;
- `similar_product_ids.# = "0684824396"` — массив `similar_product_ids` содержит строку "0684824396".
- `*.color = "red"` — внутри документа существует объект, ключ "color" в котором имеет значение "red".
- `foo = *` — в объекте существует ключ «foo».

Путь выбирает набор значений JSON для проверки, используя заданные операторы. В простейшем случае путь — это просто имя ключа. В общем случае путь задают имена ключей и подстановочные знаки, соединённые точками. Подстановочные знаки в пути могут быть следующими:

- `#` — любой элемент в массиве;
- `#N` — N-ый индекс в массиве;
- `%` — любой ключ объекта;
- `*` — любой набор элементов массивов и ключей объектов;
- `@#` — длина массива или объекта, может задаваться только в последнем компоненте пути;
- `$` — весь документ JSON в виде одного значения, может быть только единственным компонентом пути.

Результат выражения — true, когда оператор выдаёт true как минимум для одного значения, выбранного по заданному пути.

Имена ключей могут задаваться в двойных кавычках и без них. Имена ключей без кавычек не могут содержать пробелы, начинаться с цифры или совпадать с ключевым словом jsquery.

Поддерживаются следующие бинарные операторы:

- Оператор равенства: `=`;
- Операторы численного сравнения: `>`, `>=`, `<`, `<=`;
- Оператор `IN` для поиска в списке скалярных значений;
- Операторы сравнения массивов: `&&` (пересекается с), `@>` (содержит), `<@` (содержится в).
- Оператор фильтра: `~~`. Принимая данные `jsonb` в качестве левого операнда и выражение `jsquery` в качестве правого, этот оператор ищет в данных `jsonb` элементы, удовлетворяющие условию, заданному в выражении `jsquery`, и возвращает массив таких элементов, если они находятся.

Поддерживаются следующие унарные операторы:

- Оператор проверки существования: `= *`;
- Операторы проверки типа: `IS ARRAY`, `IS NUMERIC`, `IS OBJECT`, `IS STRING` и `IS BOOLEAN`.

Выражения могут быть сложными. Сложное выражение образуется набором выражений, соединённых логическими операторами (`AND`, `OR` и `NOT`) и сгруппированных скобками.

Примеры сложных выражений приведены ниже.

- `a = 1 AND (b = 2 OR c = 3) AND NOT d = 1`
- `x.% = true OR x.# = true`

Путь может включать в себя префиксные выражения (подвыражения). В этом случае путь выбирает значения JSON для проверки по заданным подвыражениям. Результаты проверки агрегируются так же, как и в простых выражениях.

- `#(a = 1 AND b = 2)` — массив содержит элемент, в котором значение ключа `a` равно 2 и значение ключа `b` равно 2
- `%($ >= 10 AND $ <= 20)` — объект содержит ключ со значением от 10 до 20

Путь также может содержать следующие подстановочные знаки, обозначающие «каждый»:

- `#:` — каждый индекс в массиве;
- `%:` — каждый ключ объекта;
- `*:` — каждый элемент массива или ключ объекта.

Рассмотрим следующий пример.

```
%.#:($ >= 0 AND $ <= 1)
```

Это условие можно прочитать как: существует минимум один ключ, для которого значением является массив чисел от 0 до 1.

Мы можем переписать этот пример в следующей форме с дополнительными скобками.

```
%(#:($ >= 0 AND $ <= 1))
```

Первый подстановочный знак `%` проверяет, что выражение в скобках истинно как минимум для одного значения в объекте. Второй подстановочный знак `#:` проверяет, что значение — массив, все элементы которого удовлетворяют условию в скобках.

Мы можем переписать этот пример без подстановочного знака `#:` следующим образом.

```
%(NOT #(NOT ($ >= 0 AND $ <= 1)) AND $ IS ARRAY)
```

В этом примере мы преобразуем утверждение, что каждый элемент массива удовлетворяет некоторому условию, в утверждение, что ни один элемент не удовлетворяет тому же условию.

Ниже приведены примеры использования путей.

- `numbers.#: IS NUMERIC` — каждый элемент массива numbers является числом.
- `*:($ IS OBJECT OR $ IS BOOLEAN)` — документ JSON представляет собой структуру вложенных объектов, на уровне листьев содержащую логические значения.
- `#:.%:($ >= 0 AND $ <= 1)` — каждый элемент массива представляет собой объект, содержащий числовые значения от 0 до 1.
- `documents.#:.% = *` — ключу documents соответствует массив из объектов, содержащих минимум один ключ.
- `%.#: ($ IS STRING)` — объект JSON содержит минимум один массив со строками.
- `#.% = true` — как минимум один элемент массива является объектом, содержащим минимум одно значение true.

Использование операторов пути и скобок требует дополнительных разъяснений. Когда операторы пути используются неоднократно, они могут ссылаться на различные значения, тогда как, используя скобки и оператор `$`, вы будете многократно обращаться к одному значению. Взгляните на следующие примеры.

- `# < 10 AND # > 20` — существует элемент меньше 10 и существует другой элемент больше 20.
- `#($ < 10 AND $ > 20)` — существует элемент, который одновременно меньше 10 и больше 20 (что невозможно).
- `#($ >= 10 AND $ <= 20)` — существует элемент между 10 и 20.
- `# >= 10 AND # <= 20` — существует элемент, больший или равный 10, и другой элемент, меньший или равный 20. Запрос может быть удовлетворён массивом без элементов между 10 и 20, например [0,30].

Те же правила применяются при поиске внутри объектов и ветвей.

Операторы проверки типа и подстановочные знаки «каждый» полезны для проверки правильности схемы документа. Оператор сопоставления JsQuery `@@` является постоянным (IMMUTABLE) и может применяться в ограничениях CHECK. Взгляните на следующий пример.

```
CREATE TABLE js (
    id serial,
    data jsonb,
    CHECK (data @@ '
        name IS STRING AND
        similar_ids.#: IS NUMERIC AND
        points.#:(x IS NUMERIC AND y IS NUMERIC)'::jsquery));
```

В этом примере ограничение-проверка контролирует данные в колонке data типа jsonb: значением ключа name должна быть строка, значением ключа similar_ids — массив чисел, значением ключа points — массив объектов, содержащих числовые значения для ключей x и y.

Другие примеры можно найти в нашей *презентации для pgconf.eu*.

F.26.3. Индексы GIN

Расширение JsQuery содержит два класса операторов для GIN, которые реализуют различные варианты оптимизации запросов.

- jsonb_path_value_ops
- jsonb_value_path_ops

В этих классах GIN документы jsonb раскладываются на элементы. Каждый элемент сопоставляется с определённым значением и своим путём. Различаются эти классы операторов тем, как в них представлены элементы, как они сравниваются и используются для оптимизации поиска.

Например, документ jsonb `{"a": [{"b": "xyz", "c": true}, 10], "d": {"e": [7, false]}}` будет разложен на следующие элементы:

- `"a".#."b"."xyz"`
- `"a".#."c".true`
- `"a".#.10`
- `"d"."e".#.7`
- `"d"."e".#.false`

Так как JsQuery не поддерживает поиск в определённом элементе массива, мы рассматриваем все элементы массива как одинаковые. Таким образом, все элементы массива помечаются одинаковым знаком `#` в пути.

Основной проблемой такого представления элементов является его размер. Так, в показанном примере ключ «a» присутствует три раза. В большом документе со множеством ветвей и длинными ключами размер такого наивного представления элементов выходит за рамки разумного. Эти два класса операторов решают эту проблему, но с некоторыми различиями.

F.26.3.1. jsonb_path_value_ops

Класс jsonb_path_value_ops представляет элемент в виде пары хеша пути и значения. Это иллюстрирует следующий псевдокод.

```
(hash(элемент_пути_1.элемент_пути_2. ... .элемент_пути_n); значение)
```

При сравнении записей хеш пути является старшей частью записи, а значение — младшей. Это определяет характеристики данного класса операторов. Так как путь хешируется и оказывается в верхней части записи, нам нужно знать полный путь к значению, чтобы использовать его для

поиска. Однако если путь известен, мы можем применять и точный поиск, и поиск значений в интервале.

F.26.3.2. jsonb_value_path_ops

Класс jsonb_value_path_ops представляет элемент в виде пары значения и фильтра Блума для пути.

```
(значение; bloom(элемент_пути_1) | bloom(элемент_пути_2) | ... | bloom(элемент_пути_n))
```

При сравнении записей значение является старшей частью записи, а фильтр Блума для пути — младшей. Это определяет характеристики данного класса операторов. Так как значение находится в старшей части, мы можем выполнять очень эффективно только поиск точного значения. Поиск значений в интервале возможен, но для этого придётся отфильтровать все другие пути, в которых будут найдены подходящие значения. Фильтр Блума по элементам пути позволяет использовать индекс для условий, содержащих в путях `%` и `*`.

F.26.3.3. Оптимизация запросов

Классы операторов JsQuery выполняют сложную оптимизацию запросов. Для разработчика или администратора очень ценно иметь возможность видеть результаты такой оптимизации. Но к сожалению, классы операторов не могут добавлять что-либо своё в вывод EXPLAIN. Именно поэтому JsQuery предоставляет следующие функции, позволяющие увидеть, как конкретный класс операторов оптимизирует заданный запрос.

- gin_debug_query_path_value(jsquery) — для jsonb_path_value_ops
- gin_debug_query_value_path(jsquery) — для jsonb_value_path_ops

Эти функции выдают текстовое представление дерева запросов, листьями в котором являются записи, искомые в индексе GIN. Следующие примеры показывают различия в результатах оптимизации запросов для разных классов операторов.

```
# SELECT gin_debug_query_path_value('x = 1 AND (*.y = 1 OR y = 2)');
 gin_debug_query_path_value
----------------------------
 x = 1 , entry 0           +

# SELECT gin_debug_query_value_path('x = 1 AND (*.y = 1 OR y = 2)');
 gin_debug_query_value_path
----------------------------
 AND                       +
   x = 1 , entry 0         +
   OR                      +
     *.y = 1 , entry 1     +
     y = 2 , entry 2       +
```

К сожалению, для jsonb ещё нет статистики. Вот почему оптимизатор JsQuery должен принимать императивные решения, выбирая условия, обрабатываемые с индексом. При этом он руководствуется предположением, что некоторые типы условий являются менее избирательными, чем другие. Таким образом, оптимизатор разделяет условия на следующие категории избирательности (перечисленные по убыванию избирательности).

1. Равняется (x = c)
2. В интервале (c1 < x < c2)
3. Не равняется (x > c)
4. Является (x is type)
5. Любое (x = *)

Оптимизатор избегает использования индекса для менее избирательных условий, когда это возможно. Например, в запросе `x = 1 AND y > 0` условие `x = 1` считается более избирательным, чем `y > 0`. Поэтому для вычисления `y > 0` индекс не используется.

```
# SELECT gin_debug_query_path_value('x = 1 AND y > 0');
```

```
 gin_debug_query_path_value
----------------------------
 x = 1 , entry 0           +
```

Решения, принимаемые оптимизатором без учёта статистики, могут быть неточными. Поэтому в JsQuery поддерживаются подсказки. Комментарии `/*-- index */` и `/*-- noindex */`, помещённые в условия, указывают оптимизатору соответственно использовать или не использовать индекс.

```
SELECT gin_debug_query_path_value('x = 1 AND y /*-- index */ > 0');
 gin_debug_query_path_value
----------------------------
 AND                       +
   x = 1 , entry 0         +
   y > 0 , entry 1         +

SELECT gin_debug_query_path_value('x /*-- noindex */ = 1 AND y > 0');
 gin_debug_query_path_value
----------------------------
  y > 0 , entry 0           +
```

F.26.4. Авторы

- Фёдор Сигаев <teodor@sigaev.ru>, Postgres Professional, Москва, Россия
- Александр Коротков <aekorotkov@gmail.com>, Postgres Professional, Москва, Россия
- Олег Бартунов <oleg@sai.msu.su>, Postgres Professional, Москва, Россия

F.26.5. Благодарности

Мы выражаем признательность спонсору разработки, компании *Wargaming.net*.

F.27. lo

Модуль `lo` поддерживает управление большими объектами (БО или LO, Large Objects, иногда BLOB, Binary Large OBjects). Он реализует тип данных `lo` и триггер `lo_manage`.

F.27.1. Обоснование

Одна из проблем драйвера JDBC (она распространяется и на драйвер ODBC) в том, что спецификация типа предполагает, что ссылки на BLOB хранятся в таблице, и если запись меняется, связанный BLOB удаляется из базы.

Но с Postgres Pro этого не происходит. Большие объекты обрабатываются как самостоятельные объекты; запись в таблице может ссылаться на большой объект по OID, но при этом на один и тот же объект могут ссылаться несколько записей таблицы, так что система не удаляет большой объект, только потому что вы меняете или удаляете такую запись.

Это не проблема для приложений, ориентированных на Postgres Pro, но стандартный код, использующий JDBC или ODBC, не будет удалять эти объекты, в результате чего они окажутся потерянными — объектами, которые никак не задействованы, а просто занимают место на диске.

Модуль `lo` позволяет решить эту проблему, добавляя триггер к таблицам, которые содержат столбцы, ссылающиеся на БО. Этот триггер по сути просто вызывает `lo_unlink`, когда вы удаляете или изменяете значение, ссылающееся на большой объект. Данный триггер предполагает, что на любой большой объект, на который ссылается контролируемый им столбец, указывает только одна ссылка!

Этот модуль также предоставляет тип данных `lo`, который просто является доменом на базе `oid`. Он может быть полезен для выделения столбцов, содержащих ссылки на большие объекты, среди столбцов, содержащих другие OID. Для использования триггера применять тип `lo` необязательно,

но этот тип может быть полезен для отслеживания столбцов в вашей базе, представляющих большие объекты, с которыми работает триггер. Кроме того, поступали сообщения, что драйвер ODBC не работает корректно, если для столбцов BLOB используется не тип `lo`.

F.27.2. Как его использовать

Пример его использования:

```
CREATE TABLE image (title TEXT, raster lo);

CREATE TRIGGER t_raster BEFORE UPDATE OR DELETE ON image
    FOR EACH ROW EXECUTE PROCEDURE lo_manage(raster);
```

Для каждого столбца, который будет содержать уникальные ссылки на большие объекты, создайте триггер `BEFORE UPDATE OR DELETE` и передайте имя столбца в качестве единственного аргумента триггера. Вы также можете сделать, чтобы триггер срабатывал только при изменениях в столбце, указав `BEFORE UPDATE OF` *`имя_столбца`*. Если вам нужно иметь в одной таблице несколько столбцов `lo`, создайте отдельный триггер для каждого (при этом обязательно нужно дать всем триггерам в одной таблице разные имена).

F.27.3. Ограничения

- При удалении таблицы, однако, всё равно будут потеряны относящиеся к ней объекты, так как триггер не будет выполняться. Этого можно избежать, выполнив перед `DROP TABLE` команду `DELETE FROM` *`таблица`*.

  То же касается и команды `TRUNCATE`.

  Если у вас уже есть, или вы подозреваете, что есть потерянные большие объекты, обратите внимание на модуль vacuumlo, который может помочь вычистить их. Имеет смысл периодически запускать vacuumlo в качестве меры, дополняющей действие триггера `lo_manage`.
- Некоторые клиентские программы могут создавать собственные таблицы, но не создавать для них соответствующие триггеры. Кроме того, и пользователи могут не создавать такие триггеры (забывая о них, либо не зная, как это сделать).

F.27.4. Автор

Питер Маунт `<peter@retep.org.uk>`

F.28. ltree

Этот модуль реализует тип данных `ltree` для представления меток данных в иерархической древовидной структуре. Он также предоставляет расширенные средства для поиска в таких деревьях.

F.28.1. Определения

Метка — это последовательность алфавитно-цифровых символов и знаков подчёркивания (например, в локали C допускаются символы `A-Za-z0-9_`). Метки должны занимать меньше 256 символов.

Примеры: `42`, `Personal_Services`

Путь метки — это последовательность из нуля или нескольких разделённых точками меток (например, `L1.L2.L3`), представляющая путь от корня иерархического дерева к конкретному узлу. Путь не может содержать больше 65535 меток.

Пример: `Top.Countries.Europe.Russia`

Модуль `ltree` предоставляет несколько типов данных:

- `ltree` хранит путь метки.
- `lquery` представляет напоминающий регулярные выражения запрос для поиска нужных значений `ltree`. Простое слово выбирает путь с этой меткой. Звёздочка (`*`) выбирает ноль или более меток. Например:

  ```
  foo         Выбирает в точности путь метки foo
  *.foo.*     Выбирает путь, содержащий метку foo
  *.foo       Выбирает путь, в котором последняя метка foo
  ```

  Звёздочке можно также добавить числовую характеристику, ограничивающую число потенциально совпадающих меток:

  ```
  *{n}        Выбирает ровно n меток
  *{n,}       Выбирает не меньше n меток
  *{n,m}      Выбирает не меньше n и не больше m меток
  *{,m}       Выбирает не больше m меток — то же самое, что и  *{0,m}
  ```

  В конце метки, отличной от звёздочки, в `lquery` можно добавить модификаторы, чтобы найти что-то сложнее, чем точное соответствие:

  ```
  @           Выбирать метки без учёта регистра, например, запросу a@ соответствует A
  *           Выбирать любую метку с данным префиксом, например запросу foo*
   соответствует foobar
  %           Выбирать начальные слова, разделённые подчёркиваниями
  ```

  Поведение модификатора `%` несколько нетривиальное. Он пытается найти соответствие по словам, а не по всей метке. Например, запросу `foo_bar%` соответствует `foo_bar_baz` но не `foo_barbaz`. В сочетании с `*`, сопоставление префикса применяется отдельно к каждому слову, например запросу `foo_bar%*` соответствует `foo1_bar2_baz`, но не `foo1_br2_baz`.

  Также вы можете записать несколько различных меток через знак `|` (обозначающий ИЛИ) для выборки любой из этих меток, либо добавить знак `!` (НЕ) в начале, чтобы выбрать все метки, не соответствующие указанным альтернативам.

  Расширенный пример `lquery`:

  ```
  Top.*{0,2}.sport*@.!football|tennis.Russ*|Spain
  a.  b.     c.      d.               e.
  ```

  Этот запрос выберет путь, который:

  a. начинается с метки `Top`

  b. и затем включает от нуля до двух меток до

  c. метки, начинающейся с префикса `sport` (без учёта регистра)

  d. затем метку, отличную от `football` и `tennis`

  e. и заканчивается меткой, которая начинается подстрокой `Russ` или в точности равна `Spain`.

- `ltxtquery` представляет подобный полнотекстовому запрос поиска подходящих значений `ltree`. Значение `ltxtquery` содержит слова, возможно с модификаторами `@`, `*`, `%` в конце; эти модификаторы имеют то же значение, что и в `lquery`. Слова можно объединять символами `&` (И), `|` (ИЛИ), `!` (НЕ) и скобками. Ключевое отличие от `lquery` состоит в том, что `ltxtquery` выбирает слова независимо от их положения в пути метки.

  Пример `ltxtquery`:

  ```
  Europe & Russia*@ & !Transportation
  ```

  Этот запрос выберет пути, содержащие метку `Europe` или любую метку с начальной подстрокой `Russia` (без учёта регистра), но не пути, содержащие метку `Transportation`.

Положение этих слов в пути не имеет значения. Кроме того, когда применяется `%`, слово может быть сопоставлено с любым другим отделённым подчёркиваниями словом в метке, вне зависимости от его положения.

Замечание: `ltxtquery` допускает пробелы между символами, а `ltree` и `lquery` — нет.

F.28.2. Операторы и функции

Для типа `ltree` определены обычные операторы сравнения `=`, `<>`, `<`, `>`, `<=`, `>=`. Сравнение сортирует пути в порядке движения по дереву, а потомки узла сортируются по тексту метки. В дополнение к ним есть и специализированные операторы, перечисленные в Таблице F.15.

Таблица F.15. Операторы `ltree`

Оператор	Возвращает	Описание
`ltree @> ltree`	`boolean`	левый аргумент является предком правого (или равен ему)?
`ltree <@ ltree`	`boolean`	левый аргумент является потомком правого (или равен ему)?
`ltree ~ lquery`	`boolean`	значение `ltree` соответствует `lquery`?
`lquery ~ ltree`	`boolean`	значение `ltree` соответствует `lquery`?
`ltree ? lquery[]`	`boolean`	значение `ltree` соответствует одному из `lquery` в массиве?
`lquery[] ? ltree`	`boolean`	значение `ltree` соответствует одному из `lquery` в массиве?
`ltree @ ltxtquery`	`boolean`	значение `ltree` соответствует `ltxtquery`?
`ltxtquery @ ltree`	`boolean`	значение `ltree` соответствует `ltxtquery`?
`ltree || ltree`	`ltree`	объединяет два пути `ltree`
`ltree || text`	`ltree`	преобразует текст в `ltree` и объединяет с путём
`text || ltree`	`ltree`	преобразует текст в `ltree` и объединяет с путём
`ltree[] @> ltree`	`boolean`	массив содержит предка `ltree`?
`ltree <@ ltree[]`	`boolean`	массив содержит предка `ltree`?
`ltree[] <@ ltree`	`boolean`	массив содержит потомка `ltree`?
`ltree @> ltree[]`	`boolean`	массив содержит потомка `ltree`?
`ltree[] ~ lquery`	`boolean`	массив содержит путь, соответствующий `lquery`?
`lquery ~ ltree[]`	`boolean`	массив содержит путь, соответствующий `lquery`?
`ltree[] ? lquery[]`	`boolean`	массив `ltree` содержит путь, соответствующий любому из `lquery`?

Оператор	Возвращает	Описание
`lquery[] ? ltree[]`	`boolean`	массив `ltree` содержит путь, соответствующий любому из `lquery`?
`ltree[] @ ltxtquery`	`boolean`	массив содержит путь, соответствующий `ltxtquery`?
`ltxtquery @ ltree[]`	`boolean`	массив содержит путь, соответствующий `ltxtquery`?
`ltree[] ?@> ltree`	`ltree`	первый элемент массива, являющийся предком `ltree`; NULL, если такого нет
`ltree[] ?<@ ltree`	`ltree`	первый элемент массива, являющийся потомком `ltree`; NULL, если такого нет
`ltree[] ?~ lquery`	`ltree`	первый элемент массива, соответствующий `lquery`; NULL, если такого нет
`ltree[] ?@ ltxtquery`	`ltree`	первый элемент массива, соответствующий `ltxtquery`; NULL, если такого нет

Операторы `<@`, `@>`, `@` и `~` имеют аналоги в виде `^<@`, `^@>`, `^@`, `^~`, которые отличатся только тем, что не используют индексы. Они полезны только для тестирования.

Доступные функции перечислены в Таблице F.16.

Таблица F.16. Функции `ltree`

Функция	Тип результата	Описание	Пример	Результат
`subltree(ltree, int start, int end)`	`ltree`	подпуть `ltree` от позиции *start* до позиции *end*-1 (начиная с 0)	`subltree( 'Top.Child1. Child2',1,2)`	`Child1`
`subpath(ltree, int offset, int len)`	`ltree`	подпуть `ltree`, начиная с позиции *offset*, длиной *len*. Если *offset* меньше нуля, подпуть начинается с этого смещения от конца пути. Если *len* меньше нуля, будет отброшено заданное число меток с конца строки.	`subpath( 'Top.Child1. Child2',0,2)`	`Top.Child1`
`subpath(ltree, int offset)`	`ltree`	подпуть `ltree`, начиная с позиции *offset* и до конца пути. Если *offset* меньше нуля, подпуть	`subpath( 'Top.Child1. Child2',1)`	`Child1.Child2`

Функция	Тип результата	Описание	Пример	Результат
		начинается с этого смещения от конца пути.		
nlevel(ltree)	integer	число меток в пути	nlevel('Top.Child1. Child2')	3
index(ltree a, ltree b)	integer	позиция первого вхождения *b* в *a*; -1, если вхождения нет	index('0.1.2.3.5.4.5.6. 8.5.6.8', '5.6')	6
index(ltree a, ltree b, int offset)	integer	позиция первого вхождения *b* в *a*, найденного от позиции *offset*; если *offset* меньше 0, поиск начинается с – *offset* меток от конца пути	index('0.1.2.3.5.4.5.6. 8.5.6.8', '5.6',-4)	9
text2ltree(text)	ltree	приводит text к типу ltree		
ltree2text(ltree)	text	приводит ltree к типу text		
lca(ltree, ltree, ...)	ltree	наибольший общий предок путей (принимается до 8 аргументов)	lca('1.2.3', '1.2.3.4.5.6')	1.2
lca(ltree[])	ltree	наибольший общий предок путей в массиве	lca(array['1.2.3'::ltree, '1.2.3.4'])	1.2

F.28.3. Индексы

ltree поддерживает несколько типов индексов, которые могут ускорить означенные операции:

- B-дерево по значениям ltree: <, <=, =, >=, >
- GiST по значениям ltree: <, <=, =, >=, >, @>, <@, @, ~, ?

  Пример создания такого индекса:

  ```
  CREATE INDEX path_gist_idx ON test USING GIST (path);
  ```

- GiST по столбцу ltree[]: ltree[] <@ ltree, ltree @> ltree[], @, ~, ?

  Пример создания такого индекса:

  ```
  CREATE INDEX path_gist_idx ON test USING GIST (array_path);
  ```

  Примечание: Индекс этого типа является неточным.

F.28.4. Пример

Для этого примера используются следующие данные (это же описание данных находится в файле contrib/ltree/ltreetest.sql в дистрибутиве исходного кода):

```
CREATE TABLE test (path ltree);
INSERT INTO test VALUES ('Top');
INSERT INTO test VALUES ('Top.Science');
INSERT INTO test VALUES ('Top.Science.Astronomy');
INSERT INTO test VALUES ('Top.Science.Astronomy.Astrophysics');
INSERT INTO test VALUES ('Top.Science.Astronomy.Cosmology');
INSERT INTO test VALUES ('Top.Hobbies');
INSERT INTO test VALUES ('Top.Hobbies.Amateurs_Astronomy');
INSERT INTO test VALUES ('Top.Collections');
INSERT INTO test VALUES ('Top.Collections.Pictures');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Stars');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Galaxies');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Astronauts');
CREATE INDEX path_gist_idx ON test USING GIST (path);
CREATE INDEX path_idx ON test USING BTREE (path);
```

В итоге мы получаем таблицу `test`, наполненную данными, представляющими следующую иерархию:

```
                      Top
                    /   |  \
             Science Hobbies Collections
                /       |              \
        Astronomy   Amateurs_Astronomy Pictures
          /  \                            |
Astrophysics  Cosmology                Astronomy
                                        /  |    \
                                  Galaxies Stars Astronauts
```

Мы можем выбрать потомки в иерархии наследования:

```
ltreetest=> SELECT path FROM test WHERE path <@ 'Top.Science';
                path
-----------------------------------
 Top.Science
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(4 rows)
```

Несколько примеров выборки по путям:

```
ltreetest=> SELECT path FROM test WHERE path ~ '*.Astronomy.*';
                     path
-----------------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Collections.Pictures.Astronomy
 Top.Collections.Pictures.Astronomy.Stars
 Top.Collections.Pictures.Astronomy.Galaxies
 Top.Collections.Pictures.Astronomy.Astronauts
(7 rows)

ltreetest=> SELECT path FROM test WHERE path ~ '*.!pictures@.*.Astronomy.*';
                path
-----------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
```

```
(3 rows)
```

Ещё несколько примеров полнотекстового поиска:

```
ltreetest=> SELECT path FROM test WHERE path @ 'Astro*% & !pictures@';
                path
----------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Hobbies.Amateurs_Astronomy
(4 rows)

ltreetest=> SELECT path FROM test WHERE path @ 'Astro* & !pictures@';
                path
----------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(3 rows)
```

Образование пути с помощью функций:

```
ltreetest=> SELECT subpath(path,0,2)||'Space'||subpath(path,2) FROM test WHERE path <@
 'Top.Science.Astronomy';
                 ?column?
------------------------------------------
 Top.Science.Space.Astronomy
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

Эту процедуру можно упростить, создав функцию SQL, вставляющую метку в определённую позицию в пути:

```
CREATE FUNCTION ins_label(ltree, int, text) RETURNS ltree
    AS 'select subpath($1,0,$2) || $3 || subpath($1,$2);'
    LANGUAGE SQL IMMUTABLE;

ltreetest=> SELECT ins_label(path,2,'Space') FROM test WHERE path <@
 'Top.Science.Astronomy';
                ins_label
------------------------------------------
 Top.Science.Space.Astronomy
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

F.28.5. Трансформации

Также имеются дополнительные расширения, реализующие трансформации типа `ltree` для языка PL/Python. Эти расширения называются `ltree_plpythonu`, `ltree_plpython2u` и `ltree_plpython3u` (соглашения об именовании, принятые для интерфейса PL/Python, описаны в Разделе 43.1). Если вы установите эти трансформации и укажете их при создании функции, значения `ltree` будут отображаться в списки Python. (Однако обратное преобразование не поддерживается.)

> **Внимание**
>
> Расширения, реализующие трансформации, настоятельно рекомендуется устанавливать в одну схему с `ltree`. Выбор какой-либо другой схемы, которая может содержать

объекты, созданные злонамеренным пользователем, чреват угрозами безопасности во время установки расширения.

F.28.6. Авторы

Разработку осуществили Фёдор Сигаев (<teodor@stack.net>) и Олег Бартунов (<oleg@sai.msu.su>). Дополнительные сведения можно найти на странице *http://www.sai.msu.su/~megera/postgres/gist/*. Авторы выражают благодарность Евгению Родичеву за полезные дискуссии. Замечания и сообщения об ошибках приветствуются.

F.29. mchar

Модуль `mchar` предоставляет дополнительные типы данных для совместимости с Microsoft SQL Server (MS SQL).

F.29.1. Обзор

Этот модуль был разработан для улучшения поддержки системы 1С:Предприятие, самой популярной в России CRM и ERP-платформы.

Он содержит реализацию типов MCHAR и MVARCHAR, которые с точностью до ошибок совместимы с типами MS SQL CHAR и VARCHAR, соответственно. Эти типы используют libicu для сравнения строк и преобразования регистра, так что они действуют одинаково в разных операционных системах.

В Postgres Pro также есть расширение citext, предоставляющие типы, подобные MCHAR. Но это расширение не воспроизводит поведение MS-SQL в отношении пробельных символов в конце строки.

Отличия от стандартных типов Postgres Pro CHAR и VARCHAR:

- Сравнение без учёта регистра
- Обработка пробельных символов в конце строки
- Эти типы всегда хранятся как двухбайтовые строки Unicode вне зависимости от кодировки базы данных.

F.29.2. Дополнительные типы

- `mchar` — аналог типа char в MS SQL
- `mvarchar` — аналог типа varchar в MS SQL

F.29.3. Функциональность типов MCHAR и MVARCHAR

- Определяется функция `length(str)`
- Определяется функция `substr(str, pos[, length])`
- Определяется оператор `||`, применяемый для конкатенации любых аргументов mchar и mvarchar
- Определяется набор операторов: `<`, `<=`, `=`, `>=` и `>` для сравнения символов без учёта регистра (LibICU)
- Определяется набор операторов: `&<`, `&<=`, `&=`, `&>=` и `&>` для сравнения с учётом регистра (LibICU)
- Неявное приведение между типами `mchar` и `mvarchar`

- Поддержка B-деревьев и хеш-индексов
- Поддержка оператора `LIKE [ESCAPE]`
- Поддержка оператора `SIMILAR TO [ESCAPE]`
- Поддержка оператора ~ (регулярные выражения POSIX)
- Поддержка индексов для оператора *LIKE*

F.29.4. Авторы

Олег Бартунов `<oleg@sai.msu.ru>`
Фёдор Сигаев `<teodor@sigaev.ru>`

F.30. online_analyze

Модуль `online_analyze` предоставляет набор функций, которые немедленно обновляют статистику после операций `INSERT`, `UPDATE`, `DELETE` или `SELECT INTO` в целевых таблицах.

F.30.1. Загрузка модуля

Чтобы использовать модуль `online_analyze`, загрузите разделяемую библиотеку:

```
LOAD 'online_analyze';
```

F.30.2. Настройка модуля

Вы можете настроить `online_analyze`, используя следующие дополнительные переменные (для них показаны значения по умолчанию):

- online_analyze.enable = on

  Включает `online_analyze`.

- online_analyze.verbose = on

  Выполняет `ANALYZE VERBOSE`.

  > **Примечание**
  >
  > Так как `verbose` является зарезервированным ключевым словом языка SQL, в SQL-запросах данный параметр нужно заключать в кавычки. Например:
  >
  > ```
  > ALTER SYSTEM SET "online_analyze.verbose" = 'off';
  > ```

- online_analyze.scale_factor = 0.1

  Процент от размера таблицы, при котором начинается немедленный анализ (этот параметр подобен autovacuum_analyze_scale_factor).

- online_analyze.threshold = 50

  Минимальное число изменений строк, после которого может начаться немедленный анализ (этот параметр подобен autovacuum_analyze_threshold).

- online_analyze.min_interval = 10000

  Минимальный интервал времени между вызовами `ANALYZE` для отдельной таблицы (в миллисекундах).

- online_analyze.table_type = "all"

Типы таблиц, для которых выполняется немедленный анализ: `all` (все), `persistent` (постоянные), `temporary` (временные), `none` (никакие).

- online_analyze.exclude_tables = ""

  Список таблиц, исключаемых из немедленного анализа.

- online_analyze.include_tables = ""

  Список таблиц, подвергаемых немедленному анализу (параметр `online_analyze.include_tables` переопределяет `online_analyze.exclude_tables`).

- online_analyze.local_tracking = off

  Включает в `online_analyze` отслеживание временных таблиц в рамках обслуживающего процесса. Когда эта переменная отключена (`off`), `online_analyze` использует для временных таблиц системную статистику по умолчанию.

- online_analyze.lower_limit = 0

  Минимальное число строк в таблиц, при котором будет срабатывать `online_analyze`.

- online_analyze.capacity_threshold = 100000

  Максимальное число временных таблиц, сохраняемых в локальном кеше.

F.30.3. Авторы

Фёдор Сигаев `<teodor@sigaev.ru>`

F.31. pageinspect

Модуль `pageinspect` предоставляет функции, позволяющие исследовать страницы баз данных на низком уровне, что бывает полезно для отладки. Все эти функции могут вызывать только суперпользователи.

F.31.1. Функции общего назначения

`get_raw_page(relname text, fork text, blkno int) returns bytea`

Функция `get_raw_page` считывает указанный блок отношения с заданным именем и возвращает копию значения `bytea`. Это позволяет получить одну согласованную во времени копию блока. В параметре *`fork`* нужно передать `'main'`, чтобы обратиться к основному слою данных, `'fsm'` — к карте свободного пространства, `'vm'` — к карте видимости, либо `'init'` — к слою инициализации.

`get_raw_page(relname text, blkno int) returns bytea`

Упрощённая версия `get_raw_page` для чтения данных из основного слоя. Синоним `get_raw_page(relname, 'main', blkno)`

`page_header(page bytea) returns record`

Функция `page_header` показывает поля, общие для всех страниц кучи и индекса Postgres Pro.

В качестве аргумента ей передаётся образ страницы, полученный в результате вызова `get_raw_page`. Например:

```
test=# SELECT * FROM page_header(get_raw_page('pg_class', 0));
    lsn    | checksum | flags  | lower | upper | special | pagesize | version |
 prune_xid
-----------+----------+--------+-------+-------+---------+----------+---------
+-----------
```

```
 0/24A1B50 |        0 |      1 |   232 |   368 |    8192 |     8192 |       4 |
    0
```

Возвращаемые столбцы соответствуют полям в структуре `PageHeaderData`. За подробностями обратитесь к `src/include/storage/bufpage.h`.

Поле `checksum` содержит контрольную сумму, сохранённую для страницы. Эта сумма может быть неверной при повреждении страницы. Если в данном экземпляре кластера контрольные суммы отключены, это значение не имеет смысла.

`page_checksum(page bytea, blkno int4) returns smallint`

Функция `page_checksum` вычисляет контрольную сумму страницы, которая должна была бы находиться в заданном блоке.

В качестве аргумента ей передаётся образ страницы, полученный в результате вызова `get_raw_page`. Например:

```
test=# SELECT page_checksum(get_raw_page('pg_class', 0), 0);
 page_checksum
---------------
         13443
```

Заметьте, что вычисление контрольной суммы зависит от номера блока, поэтому обеим функциям нужно передавать одинаковые номера (за исключением случаев эзотерической отладки).

Контрольную сумму, вычисленную этой функцией, можно сравнить с полем `checksum` результата функции `page_header`. Если контрольные суммы для текущего экземпляра БД включены, эти значения должны быть равны.

`fsm_page_contents(page bytea) returns text`

Функция `fsm_page_contents` показывает внутреннюю структуру узла на странице FSM. Например:

```
test=# SELECT fsm_page_contents(get_raw_page('pg_class', 'fsm', 0));
```

Данный запрос выводит несколько текстовых строк, по одной строке для каждого узла двоичного дерева на заданной странице. Нулевые узлы при этом пропускаются. Также выводится так называемый указатель «вперёд», который указывает на следующий слот, получаемый с этой страницы.

F.31.2. Функции для исследования кучи

`heap_page_items(page bytea) returns setof record`

Функция `heap_page_items` показывает все указатели линейных блоков на странице кучи. Для используемых блоков также выводятся заголовки кортежей. При этом показываются все кортежи, независимо от того, были ли видны они в снимке MVCC в момент копирования исходной страницы.

В качестве аргумента ей передаётся образ страницы кучи, полученный в результате вызова `get_raw_page`. Например:

```
test=# SELECT * FROM heap_page_items(get_raw_page('pg_class', 0));
```

Описание возвращаемых полей можно найти в `src/include/storage/itemid.h` и `src/include/access/htup_details.h`.

`tuple_data_split(rel_oid oid, t_data bytea, t_infomask integer, t_infomask2 integer, t_bits text [, do_detoast bool]) returns bytea[]`

Функция `tuple_data_split` разделяет данные кортежей на атрибуты так, как это происходит внутри сервера.

```
test=# SELECT tuple_data_split('pg_class'::regclass, t_data, t_infomask,
 t_infomask2, t_bits) FROM heap_page_items(get_raw_page('pg_class', 0));
```

В качестве аргументов этой функции должны передаваться атрибуты, возвращаемые функцией `heap_page_items`.

Если параметр *`do_detoast`* равен `true`, полученные атрибуты будут распакованы по мере необходимости. Если он не задан, подразумевается `false`.

`heap_page_item_attrs(page bytea, rel_oid regclass [, do_detoast bool]) returns setof record`

Функция `heap_page_item_attrs` похожа на `heap_page_items`, но возвращает неструктурированное содержимое кортежа в виде массива атрибутов, которые могут быть распакованы, если установлен флаг *`do_detoast`* (по умолчанию они не распаковываются).

В качестве аргумента ей передаётся образ страницы кучи, полученный в результате вызова `get_raw_page`. Например:

```
test=# SELECT * FROM heap_page_item_attrs(get_raw_page('pg_class', 0),
 'pg_class'::regclass);
```

F.31.3. Функции для индексов-B-деревьев

`bt_metap(relname text) returns record`

Функция `bt_metap` возвращает информацию о метастранице индекса-B-дерева. Например:

```
test=# SELECT * FROM bt_metap('pg_cast_oid_index');
-[ RECORD 1 ]-----
magic     | 340322
version   | 2
root      | 1
level     | 0
fastroot  | 1
fastlevel | 0
```

`bt_page_stats(relname text, blkno int) returns record`

`bt_page_stats` возвращает сводную информацию по единичным страницам B-дерева. Например:

```
test=# SELECT * FROM bt_page_stats('pg_cast_oid_index', 1);
-[ RECORD 1 ]-+-----
blkno         | 1
type          | l
live_items    | 256
dead_items    | 0
avg_item_size | 12
page_size     | 8192
free_size     | 4056
btpo_prev     | 0
btpo_next     | 0
btpo          | 0
btpo_flags    | 3
```

`bt_page_items(relname text, blkno int) returns setof record`

`bt_page_items` возвращает детализированную информацию обо всех элементах на странице B-дерева. Например:

```
test=# SELECT * FROM bt_page_items('pg_cast_oid_index', 1);
 itemoffset |  ctid   | itemlen | nulls | vars |    data
```

```
------------+---------+---------+-------+------+-------------
          1 | (0,1)   |      12 | f     | f    | 23 27 00 00
          2 | (0,2)   |      12 | f     | f    | 24 27 00 00
          3 | (0,3)   |      12 | f     | f    | 25 27 00 00
          4 | (0,4)   |      12 | f     | f    | 26 27 00 00
          5 | (0,5)   |      12 | f     | f    | 27 27 00 00
          6 | (0,6)   |      12 | f     | f    | 28 27 00 00
          7 | (0,7)   |      12 | f     | f    | 29 27 00 00
          8 | (0,8)   |      12 | f     | f    | 2a 27 00 00
```

На странице уровня листьев B-дерева, `ctid` указывает на кортеж в куче. На внутренней странице часть `ctid`, содержащая номер блока, указывает на другую страницу в самом индексе, а часть смещения (второе число) игнорируется и обычно равняется 1.

Заметьте, что первый элемент в любой, кроме самой правой, странице (то есть в любой странице с ненулевым значением в поле `btpo_next`) представляет собой «верхний ключ», то есть его поле `data` задаёт верхнюю границу для всех элементов, находящихся на странице, а поле `ctid` лишено смысла. Кроме того, на внутренних страницах первый действительный элемент данных (первый элемент после верхнего ключа) представляет элемент «минус бесконечность», без фактического значения в поле `data`. Однако такой элемент содержит в своём поле `ctid` корректную ссылку на данные.

`bt_page_items(page bytea) returns setof record`

Также можно передать функции `bt_page_items` страницу в виде значения `bytea`. Образ страницы для передачи в аргументе можно получить в результате вызова `get_raw_page`. Таким образом, последний пример можно также переписать так:

```
test=# SELECT * FROM bt_page_items(get_raw_page('pg_cast_oid_index', 1));
 itemoffset |  ctid   | itemlen | nulls | vars |    data
------------+---------+---------+-------+------+-------------
          1 | (0,1)   |      12 | f     | f    | 23 27 00 00
          2 | (0,2)   |      12 | f     | f    | 24 27 00 00
          3 | (0,3)   |      12 | f     | f    | 25 27 00 00
          4 | (0,4)   |      12 | f     | f    | 26 27 00 00
          5 | (0,5)   |      12 | f     | f    | 27 27 00 00
          6 | (0,6)   |      12 | f     | f    | 28 27 00 00
          7 | (0,7)   |      12 | f     | f    | 29 27 00 00
          8 | (0,8)   |      12 | f     | f    | 2a 27 00 00
```

Все остальные детали те же, что и с предыдущим вариантом вызова.

F.31.4. Функции для индексов BRIN

`brin_page_type(page bytea) returns text`

Функция `brin_page_type` возвращает тип страницы для заданной страницы индекса BRIN или выдаёт ошибку, если эта страница не является корректной страницей индекса BRIN. Например:

```
test=# SELECT brin_page_type(get_raw_page('brinidx', 0));
 brin_page_type
----------------
 meta
```

`brin_metapage_info(page bytea) returns record`

Функция `brin_metapage_info` возвращает разнообразные сведения о метастранице индекса BRIN. Например:

```
test=# SELECT * FROM brin_metapage_info(get_raw_page('brinidx', 0));
   magic    | version | pagesperrange | lastrevmappage
------------+---------+---------------+---------------
```

```
 0xA8109CFA |       1 |             4 |             2
```

`brin_revmap_data(page bytea) returns setof tid`

Функция `brin_revmap_data` выдаёт список идентификаторов кортежей со страницы сопоставлений зон индекса BRIN. Например:

```
test=# SELECT * FROM brin_revmap_data(get_raw_page('brinidx', 2)) LIMIT 5;
  pages
---------
 (6,137)
 (6,138)
 (6,139)
 (6,140)
 (6,141)
```

`brin_page_items(page bytea, index oid) returns setof record`

Функция `brin_page_items` выдаёт содержимое, сохранённое в странице данных BRIN. Например:

```
test=# SELECT * FROM brin_page_items(get_raw_page('brinidx', 5),
                                     'brinidx')
       ORDER BY blknum, attnum LIMIT 6;
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |    value
------------+--------+--------+----------+----------+-------------+-------------
        137 |      0 |      1 | t        | f        | f           |
        137 |      0 |      2 | f        | f        | f           | {1 .. 88}
        138 |      4 |      1 | t        | f        | f           |
        138 |      4 |      2 | f        | f        | f           | {89 .. 176}
        139 |      8 |      1 | t        | f        | f           |
        139 |      8 |      2 | f        | f        | f           | {177 .. 264}
```

Возвращаемые столбцы соответствуют полям в структурах `BrinMemTuple` и `BrinValues`. Подробнее они описаны в `src/include/access/brin_tuple.h`.

F.31.5. Функции для индексов GIN

`gin_metapage_info(page bytea) returns record`

Функция `gin_metapage_info` выдаёт информацию о метастранице индекса GIN. Например:

```
test=# SELECT * FROM gin_metapage_info(get_raw_page('gin_index', 0));
-[ RECORD 1 ]----+-----------
pending_head     | 4294967295
pending_tail     | 4294967295
tail_free_size   | 0
n_pending_pages  | 0
n_pending_tuples | 0
n_total_pages    | 7
n_entry_pages    | 6
n_data_pages     | 0
n_entries        | 693
version          | 2
```

`gin_page_opaque_info(page bytea) returns record`

Функция `gin_page_opaque_info` выдаёт информацию из непрозрачной области индекса GIN, например, тип страницы. Например:

```
test=# SELECT * FROM gin_page_opaque_info(get_raw_page('gin_index', 2));
 rightlink | maxoff |         flags
-----------+--------+-----------------------
```

```
          5 |      0 | {data,leaf,compressed}
(1 row)
```

gin_leafpage_items(page bytea) returns setof record

Функция gin_leafpage_items выдаёт информацию о данных, хранящихся в странице индекса GIN на уровне листьев. Например:

```
test=# SELECT first_tid, nbytes, tids[0:5] AS some_tids
        FROM gin_leafpage_items(get_raw_page('gin_test_idx', 2));
 first_tid | nbytes |                          some_tids
-----------+--------+-------------------------------------------------------------
 (8,41)    |    244 | {"(8,41)","(8,43)","(8,44)","(8,45)","(8,46)"}
 (10,45)   |    248 | {"(10,45)","(10,46)","(10,47)","(10,48)","(10,49)"}
 (12,52)   |    248 | {"(12,52)","(12,53)","(12,54)","(12,55)","(12,56)"}
 (14,59)   |    320 | {"(14,59)","(14,60)","(14,61)","(14,62)","(14,63)"}
 (167,16)  |    376 | {"(167,16)","(167,17)","(167,18)","(167,19)","(167,20)"}
 (170,30)  |    376 | {"(170,30)","(170,31)","(170,32)","(170,33)","(170,34)"}
 (173,44)  |    197 | {"(173,44)","(173,45)","(173,46)","(173,47)","(173,48)"}
(7 rows)
```

F.31.6. Функции для хеш-индексов

hash_page_type(page bytea) returns text

Функция hash_page_type возвращает тип страницы для заданной страницы хеш-индекса. Например:

```
test=# SELECT hash_page_type(get_raw_page('con_hash_index', 0));
 hash_page_type
----------------
 metapage
```

hash_page_stats(page bytea) returns setof record

Функция hash_page_stats возвращает информацию о странице группы или переполнения хеш-индекса. Например:

```
test=# SELECT * FROM hash_page_stats(get_raw_page('con_hash_index', 1));
-[ RECORD 1 ]---+-----------
live_items      | 407
dead_items      | 0
page_size       | 8192
free_size       | 8
hasho_prevblkno | 4096
hasho_nextblkno | 8474
hasho_bucket    | 0
hasho_flag      | 66
hasho_page_id   | 65408
```

hash_page_items(page bytea) returns setof record

Функция hash_page_items возвращает информацию о данных, хранящихся на странице группы или переполнения хеш-индекса. Например:

```
test=# SELECT * FROM hash_page_items(get_raw_page('con_hash_index', 1)) LIMIT 5;
 itemoffset |   ctid    |    data
------------+-----------+------------
          1 | (899,77)  | 1053474816
          2 | (897,29)  | 1053474816
          3 | (894,207) | 1053474816
          4 | (892,159) | 1053474816
```

```
          5 | (890,111) | 1053474816
```

```
hash_bitmap_info(index oid, blkno int) returns record
```

Функция `hash_bitmap_info` показывает состояние бита в странице битовой карты для определённой страницы переполнения хеш-индекса. Например:

```
test=# SELECT * FROM hash_bitmap_info('con_hash_index', 2052);
 bitmapblkno | bitmapbit | bitstatus
-------------+-----------+-----------
          65 |         3 | t
```

```
hash_metapage_info(page bytea) returns record
```

`hash_metapage_info` возвращает информацию, хранящуюся в метастранице хеш-индекса. Например:

```
test=# SELECT magic, version, ntuples, ffactor, bsize, bmsize, bmshift,
test-#      maxbucket, highmask, lowmask, ovflpoint, firstfree, nmaps, procid,
test-#      regexp_replace(spares::text, '(,0)*}', '}') as spares,
test-#      regexp_replace(mapp::text, '(,0)*}', '}') as mapp
test-# FROM hash_metapage_info(get_raw_page('con_hash_index', 0));
-[ RECORD
 1 ]----------------------------------------------------------------------------
magic     | 105121344
version   | 4
ntuples   | 500500
ffactor   | 40
bsize     | 8152
bmsize    | 4096
bmshift   | 15
maxbucket | 12512
highmask  | 16383
lowmask   | 8191
ovflpoint | 28
firstfree | 1204
nmaps     | 1
procid    | 450
spares    |
 {0,0,0,0,0,0,1,1,1,1,1,1,1,1,3,4,4,4,45,55,58,59,508,567,628,704,1193,1202,1204}
mapp      | {65}
```

F.32. passwordcheck

Модуль `passwordcheck` проверяет пароли пользователей, задаваемые командами CREATE ROLE и ALTER ROLE. Если пароль признаётся слишком слабым, он не принимается и команда завершается ошибкой.

Чтобы задействовать этот модуль, добавьте строку `'$libdir/passwordcheck'` в переменную shared_preload_libraries в `postgresql.conf`, а затем перезапустите сервер.

> **Внимание**
>
> Чтобы незашифрованные пароли не передавались по сети, не записывались в журнал сервера и не стали каким-либо образом известны администратору баз данных, Postgres Pro позволяет пользователю передавать предварительно зашифрованные пароли. Используя это, клиентские программы могут шифровать пароль, прежде чем передавать его серверу.
>
> Это ограничивает полезность модуля `passwordcheck`, так как в этом случае можно только попытаться угадать пароль. Поэтому использовать `passwordcheck` не рекомендуется, когда

требуется высокий уровень безопасности. Более безопасно будет применить внешний вариант проверки подлинности, например GSSAPI (см. Главу 19), а не использовать пароли, хранящиеся в базе данных.

F.33. pg_buffercache

Модуль pg_buffercache даёт возможность понять, что происходит в общем кеше буферов в реальном времени.

Этот модуль предоставляет функцию на C pg_buffercache_pages, возвращающую набор записей, плюс представление pg_buffercache, которое является удобной обёрткой этой функции.

По умолчанию его использование разрешено только суперпользователям и членам роли pg_monitor. Дать доступ другим можно с помощью GRANT.

F.33.1. Представление pg_buffercache

Определения столбцов, содержащихся в представлении, показаны в Таблице F.17.

Таблица F.17. Столбцы pg_buffercache

Имя	Тип	Ссылки	Описание
bufferid	integer		ID, в диапазоне 1..shared_buffers
relfilenode	oid	pg_class.relfilenode	Номер файлового узла для отношения
reltablespace	oid	pg_tablespace.oid	OID табличного пространства, содержащего отношение
reldatabase	oid	pg_database.oid	OID базы данных, содержащей отношение
relforknumber	smallint		Номер слоя в отношении; см. include/common/relpath.h
relblocknumber	bigint		Номер страницы в отношении
isdirty	boolean		Страница загрязнена?
usagecount	smallint		Счётчик обращений по часовой стрелке
pinning_backends	integer		Число обслуживающих процессов, закрепивших этот буфер

Для каждого буфера в общем кеше выдаётся одна строка. Для неиспользуемых буферов все поля равны NULL, за исключением bufferid. Общие системные каталоги показываются как относящиеся к базе данных под номером 0.

Так как кеш используется совместно всеми базами данных, обычно в нём находятся и страницы из отношений, не принадлежащих текущей базе данных. Это означает, что для некоторых

строк при соединении с pg_class не найдутся соответствующие строки, либо соединение будет некорректным. Если вы хотите выполнить соединение с pg_class, будет правильным ограничить соединение строками, в которых reldatabase содержит OID текущей базы данных или ноль.

Для копирования данных состояния, которые будут отображаться, блокировки в менеджере буферов не устанавливаются. Поэтому обращения к представлению pg_buffercache меньше сказываются на обычной активности буферов, но набор результатов, полученный для всех буферов, в целом может оказаться несогласованным. Однако внутри каждого отдельного буфера согласованность информации гарантируется.

F.33.2. Пример вывода

```
regression=# SELECT n.nspname, c.relname, count(*) AS buffers
             FROM pg_buffercache b JOIN pg_class c
             ON b.relfilenode = pg_relation_filenode(c.oid) AND
                b.reldatabase IN (0, (SELECT oid FROM pg_database
                                      WHERE datname = current_database()))
             JOIN pg_namespace n ON n.oid = c.relnamespace
             GROUP BY n.nspname, c.relname
             ORDER BY 3 DESC
             LIMIT 10;

  nspname   |        relname          | buffers
------------+-------------------------+---------
 public     | delete_test_table       |     593
 public     | delete_test_table_pkey  |     494
 pg_catalog | pg_attribute            |     472
 public     | quad_poly_tbl           |     353
 public     | tenk2                   |     349
 public     | tenk1                   |     349
 public     | gin_test_idx            |     306
 pg_catalog | pg_largeobject          |     206
 public     | gin_test_tbl            |     188
 public     | spgist_text_tbl         |     182
(10 rows)
```

F.33.3. Авторы

Марк Кирквуд <markir@paradise.net.nz>

Предложения по конструкции: Нейл Конвей <neilc@samurai.com>

Советы по отладке: Том Лейн <tgl@sss.pgh.pa.us>

F.34. pgcrypto

Модуль pgcrypto предоставляет криптографические функции для Postgres Pro.

F.34.1. Стандартные функции хеширования

F.34.1.1. digest()

```
digest(data text, type text) returns bytea
digest(data bytea, type text) returns bytea
```

Вычисляет двоичный хеш данных (*data*). Параметр *type* выбирает используемый алгоритм. Поддерживаются стандартные алгоритмы: md5, sha1, sha224, sha256, sha384 и sha512. Если модуль pgcrypto собирался с OpenSSL, становятся доступны и другие алгоритмы, как описано в Таблице F.21.

Если вы хотите получить дайджест в виде шестнадцатеричной строки, примените `encode()` к результату. Например:

```
CREATE OR REPLACE FUNCTION sha1(bytea) returns text AS $$
    SELECT encode(digest($1, 'sha1'), 'hex')
$$ LANGUAGE SQL STRICT IMMUTABLE;
```

F.34.1.2. `hmac()`

```
hmac(data text, key text, type text) returns bytea
hmac(data bytea, key bytea, type text) returns bytea
```

Вычисляет имитовставку на основе хеша для данных *`data`* с ключом *`key`*. Параметр *`type`* имеет то же значение, что и для `digest()`.

Эта функция похожа на `digest()`, но вычислить хеш с ней можно, только зная ключ. Это защищает от сценария подмены данных и хеша вместе с ними.

Если размер ключа больше размера блока хеша, он сначала хешируется, а затем используется в качестве ключа хеширования данных.

F.34.2. Функции хеширования пароля

Функции `crypt()` и `gen_salt()` разработаны специально для хеширования паролей. Функция `crypt()` выполняет хеширование, а `gen_salt()` подготавливает параметры алгоритма для неё.

Алгоритмы в `crypt()` отличаются от обычных алгоритмов хеширования MD5 и SHA1 в следующих аспектах:

1. Они медленные. Так как объём данных невелик, это единственный способ усложнить перебор паролей.
2. Они используют случайное значение, называемое *солью*, чтобы у пользователей с одинаковыми паролями зашифрованные пароли оказывались разными. Это также обеспечивает дополнительную защиту от получения обратного алгоритма.
3. Они включают в результат тип алгоритма, что допускает сосуществование паролей, хешированных разными алгоритмами.
4. Некоторые из них являются адаптируемыми — то есть с ростом производительности компьютеров эти алгоритмы можно настроить так, чтобы они стали медленнее, при этом сохраняя совместимость с существующими паролями.

В Таблице F.18 перечислены алгоритмы, поддерживаемые функцией `crypt()`.

Таблица F.18. Алгоритмы, которые поддерживает `crypt()`

Алгоритм	Макс. длина пароля	Адаптивный?	Размер соли (бит)	Размер результата	Описание
`bf`	72	да	128	60	На базе Blowfish, вариация 2a
`md5`	без ограничений	нет	48	34	crypt на базе MD5
`xdes`	8	да	24	20	Расширенный DES
`des`	8	нет	12	13	Изначальный crypt из UNIX

F.34.2.1. `crypt()`

```
crypt(password text, salt text) returns text
```

Вычисляет хеш пароля (*password*) в стиле crypt(3). Для сохранения нового пароля необходимо вызвать `gen_salt()`, чтобы сгенерировать новое значение соли (*salt*). Для проверки пароля нужно передать сохранённое значение хеша в параметре *salt* и проверить, соответствует ли результат сохранённому значению.

Пример установки нового пароля:

```
UPDATE ... SET pswhash = crypt('new password', gen_salt('md5'));
```

Пример проверки пароля:

```
SELECT (pswhash = crypt('entered password', pswhash)) AS pswmatch FROM ... ;
```

Этот запрос возвращает `true`, если введённый пароль правильный.

F.34.2.2. `gen_salt()`

```
gen_salt(type text [, iter_count integer ]) returns text
```

Вычисляет новое случайное значение соли для функции `crypt()`. Строка соли также говорит `crypt()`, какой алгоритм использовать.

Параметр *type* задаёт алгоритм хеширования. Принимаются следующие варианты: `des`, `xdes`, `md5` и `bf`.

Параметр *iter_count* позволяет пользователю указать счётчик итераций для алгоритма, который его принимает. Чем больше это число, тем больше времени уйдёт на вычисление хеша пароля, а значит, тем больше времени понадобится, чтобы взломать его. Хотя со слишком большим значением время вычисления хеша может вырасти до нескольких лет — это вряд ли практично. Когда параметр *iter_count* опускается, применяется количество итераций по умолчанию. Множество допустимых значений для *iter_count* зависит от алгоритма, как показано в Таблице F.19.

Таблица F.19. Счётчики итераций для `crypt()`

Алгоритм	По умолчанию	Мин.	Макс.
`xdes`	725	1	16777215
`bf`	6	4	31

Для `xdes` есть дополнительное ограничение: счётчик итераций должен быть нечётным.

При выборе подходящего числа итераций учтите, что оригинальный алгоритм DES crypt был рассчитан так, чтобы выдавать 4 хеша в секунду на компьютерах того времени. Если за секунду будет вычисляться меньше 4 хешей, скорее всего, возникнут определённые неудобства при пользовании. С другой стороны, скорость больше, чем 100 хешей в секунду, вероятно, будет слишком высокой.

В Таблице F.20 дана сводка относительной скорости различных алгоритмов хеширования. В таблице показано, сколько времени уйдёт на перебор всех комбинацией символов в восьмисимвольном пароле, в предположении, что пароль содержит только буквы в нижнем регистре, либо буквы в верхнем и нижнем регистре, а также цифры. В строках `crypt-bf` числа после косой черты показывают значение параметра *iter_count* функции `gen_salt`.

Таблица F.20. Скорости алгоритмов хеширования

Алгоритм	Хешей/сек.	Для `[a-z]`	Для `[A-Za-z0-9]`	Длительность относительно `md5`
`crypt-bf/8`	1792	4 года	3927 лет	100k

Алгоритм	**Хешей/сек.**	**Для** `[a-z]`	**Для** `[A-Za-z0-9]`	**Длительность относительно** `md5`
`crypt-bf/7`	3648	2 года	1929 лет	50k
`crypt-bf/6`	7168	1 год	982 лет	25k
`crypt-bf/5`	13504	188 дней	521 лет	12.5k
`crypt-md5`	171584	15 дней	41 год	1k
`crypt-des`	23221568	157.5 минут	108 дней	7
`sha1`	37774272	90 минут	68 дней	4
`md5` (хеш)	150085504	22.5 минут	17 дней	1

Замечания:

- Для расчётов использовался процессор Intel Mobile Core i3.
- Показатели алгоритмов `crypt-des` и `crypt-md5` взяты из вывода теста программы John the Ripper v1.6.38.
- Показатели `md5` получены программой mdcrack 1.2.
- Показатели `sha1` получены программой lcrack-20031130-beta.
- Показатели `crypt-bf` получены простой программой, обрабатывающей в цикле 1000 паролей из 8-символов. Таким способом можно показать скорость с разным числом итераций. Для справки: `john -test` показывает 13506 циклов/с для `crypt-bf/5`. (Это очень небольшое различие в результатах согласуется с тем фактом, что реализация `crypt-bf` в `pgcrypto` не отличается от применяемой в программе John the Ripper.)

Заметьте, что вариант «перепробовать все комбинации» не вполне реалистичен. Обычно перебор паролей производится с применением словарей, которые содержат и обычные слова, и их различные видоизменения. Поэтому даже похожие на слова пароли обычно можно подобрать быстрее, чем за указанное время, тогда как 6-символьный несловесный пароль может избежать взлома. А может и не избежать.

F.34.3. Функции шифрования на базе PGP

Функции, описанные здесь, реализуют часть стандарта OpenPGP (RFC 4880), относящуюся к шифрованию. Они поддерживают шифрование как с симметричным, так и с закрытым ключом.

Зашифрованное PGP сообщение состоит из 2 частей или *пакетов*:

- Пакет, содержащий сеансовый ключ — либо симметричный, либо открытый (в зашифрованном виде).
- Пакет, содержащий данные, зашифрованные сеансовым ключом.

При шифровании с симметричным ключом (то есть, паролем):

1. Заданный пароль хешируется по алгоритму String2Key (S2K). Этот алгоритм подобен алгоритмам `crypt()` — специально замедлен и добавляет случайную соль — но на выход выдаёт двоичный ключ полной длины.
2. Если требуется отдельный сеансовый ключ, генерируется новый случайный ключ. В противном случае в качестве сеансового будет использоваться непосредственно ключ S2K.
3. Когда используется непосредственно ключ S2K, в пакет сеансового ключа помещаются только параметры S2K. В противном случае сеансовый ключ шифруется ключом S2K и результат помещается в пакет сеансового ключа.

При шифровании с открытым ключом:

1. Генерируется новый случайный сеансовый ключ.
2. Он зашифровывается открытым ключом и помещается в пакет сеансового ключа.

В любом случае данные, которые должны быть зашифрованы, обрабатываются так:

1. Необязательная подготовка данных: сжатие, перекодировка в UTF-8 и/или преобразование концов строк.
2. Перед данными добавляется блок случайных байт. Это равносильно использованию случайного вектора инициализации.
3. В конце добавляется хеш SHA1 случайного префикса и данных.
4. Всё это шифруется сеансовым ключом и помещается в пакет данных.

F.34.3.1. pgp_sym_encrypt()

```
pgp_sym_encrypt(data text, psw text [, options text ]) returns bytea
pgp_sym_encrypt_bytea(data bytea, psw text [, options text ]) returns bytea
```

Шифрует данные (*data*) симметричным ключом PGP *psw*. В *options* могут передаваться криптографические параметры, описанные ниже.

F.34.3.2. pgp_sym_decrypt()

```
pgp_sym_decrypt(msg bytea, psw text [, options text ]) returns text
pgp_sym_decrypt_bytea(msg bytea, psw text [, options text ]) returns bytea
```

Расшифровывает сообщение, зашифрованное симметричным ключом PGP.

Расшифровывать данные `bytea` функцией `pgp_sym_decrypt` запрещено. Это ограничение введено, чтобы не допустить вывода некорректных символьных данных. Расшифровывать изначально текстовые данные с помощью `pgp_sym_decrypt_bytea` можно без ограничений.

Аргумент *options* может содержать криптографические параметры, описанные ниже.

F.34.3.3. pgp_pub_encrypt()

```
pgp_pub_encrypt(data text, key bytea [, options text ]) returns bytea
pgp_pub_encrypt_bytea(data bytea, key bytea [, options text ]) returns bytea
```

Зашифровывает данные (*data*) открытым ключом PGP (*key*). Если передать этой функции закрытый ключ, она выдаст ошибку.

Аргумент *options* может содержать криптографические параметры, описанные ниже.

F.34.3.4. pgp_pub_decrypt()

```
pgp_pub_decrypt(msg bytea, key bytea [, psw text [, options text ]]) returns text
pgp_pub_decrypt_bytea(msg bytea, key bytea [, psw text [, options text ]]) returns
 bytea
```

Расшифровывает сообщение, зашифрованное открытым ключом. В *key* должен передаваться закрытый ключ, соответствующий открытому ключу, применяющемуся при шифровании. Если секретный ключ защищён паролем, этот пароль нужно передать в параметре *psw*. Если пароля нет, но необходимо передать криптографические параметры, вы должны передать пустой пароль.

Расшифровывать данные `bytea` функцией `pgp_pub_decrypt` запрещено. Это ограничение введено, чтобы не допустить вывода недопустимых символьных данных. Расшифровывать изначально текстовые данные с помощью `pgp_pub_decrypt_bytea` можно без ограничений.

Аргумент *options* может содержать криптографические параметры, описанные ниже.

F.34.3.5. pgp_key_id()

```
pgp_key_id(bytea) returns text
```

pgp_key_id извлекает идентификатор ключа из открытого или закрытого ключа PGP. Она также может выдать идентификатор ключа, которым были зашифрованы данные, если ей передаётся зашифрованное сообщение.

Она может выдать два специальных идентификатора ключа:

- SYMKEY

  Сообщение зашифровано симметричным ключом.

- ANYKEY

  Сообщение зашифровано открытом ключом, но идентификатор ключа был удалён. Это означает, что вам надо будет перепробовать ключи, чтобы подобрать подходящий. Сама библиотека pgcrypto не генерирует такие сообщения.

Заметьте, что разные ключи могут иметь одинаковый идентификатор. Это редкое, но не невероятное явление. В таком случае клиентское приложение должно пытаться расшифровать данные с каждым ключом, пока не найдёт подходящий — примерно так же, как и с ANYKEY.

F.34.3.6. armor(), dearmor()

```
armor(data bytea [ , keys text[], values text[] ]) returns text
dearmor(data text) returns bytea
```

Эти функции переводят двоичные данные в/из формата PGP «ASCII Armor», по сути представляющий собой кодировку Base64 с контрольными суммами и дополнительным форматированием.

Если задаются массивы *keys* и *values*, для каждой пары ключ/значения в формат Armor добавляется *заголовок Armor*. Оба массива должны быть одномерными и иметь одинаковую длину. Задаваемые ключи и значения могут содержать только символы ASCII.

F.34.3.7. pgp_armor_headers

```
pgp_armor_headers(data text, key out text, value out text) returns setof record
```

Функция pgp_armor_headers() извлекает заголовки Armor из параметра *data*. Она возвращает набор строк с двумя столбцами, key и value. Если в ключах или значениях оказываются символы не ASCII, они воспринимаются как UTF-8.

F.34.3.8. Параметры функций PGP

Имена параметров подобны принятым в GnuPG. Значение параметра должно задаваться после знака равно; друг от друга параметры отделяются запятыми. Например:

```
pgp_sym_encrypt(data, psw, 'compress-algo=1, cipher-algo=aes256')
```

Все эти параметры, кроме convert-crlf, применяются только к функциям шифрования. Функции расшифровывания получают параметры из данных PGP.

Вероятно, самые интересные параметры — это compress-algo и unicode-mode. Остальные должны иметь достаточно адекватные значения по умолчанию.

F.34.3.8.1. cipher-algo

Выбирает алгоритм шифрования.

Значения: bf, aes128, aes192, aes256 (только OpenSSL: `3des`, `cast5`)
По умолчанию: aes128
Применим к: pgp_sym_encrypt, pgp_pub_encrypt

F.34.3.8.2. compress-algo

Выбирает алгоритм сжатия. Принимается, только если Postgres Pro собран с zlib.

Значения:
0 — без сжатия
1 — сжатие ZIP
2 — сжатие ZLIB (= ZIP плюс метаданные и CRC блоков)
По умолчанию: 0
Применим к: pgp_sym_encrypt, pgp_pub_encrypt

F.34.3.8.3. compress-level

Определяет уровень сжатия. Чем больше уровень, тем меньшего объёма результат, но длительнее процесс. Значение 0 отключает сжатие.

Значения: 0, 1-9
По умолчанию: 6
Применим к: pgp_sym_encrypt, pgp_pub_encrypt

F.34.3.8.4. convert-crlf

Определяет, преобразовывать ли `\n` в `\r\n` при шифровании и `\r\n` в `\n` при дешифровании. В RFC 4880 требуется, чтобы текстовые данные хранились с переводами строк в виде `\r\n`. Воспользуйтесь этим параметром, чтобы поведение полностью соответствовало RFC.

Значения: 0, 1
По умолчанию: 0
Применим к: pgp_sym_encrypt, pgp_pub_encrypt, pgp_sym_decrypt, pgp_pub_decrypt

F.34.3.8.5. disable-mdc

Не защищать данные хешем SHA-1. Единственная разумная причина использовать этот параметр — добиться совместимости с древними программами PGP, вышедшими до того, как в RFC 4880 была предусмотрена защита пакетов с SHA-1. Все последние реализации с gnupg.org и pgp.com прекрасно поддерживают это.

Значения: 0, 1
По умолчанию: 0
Применим к: pgp_sym_encrypt, pgp_pub_encrypt

F.34.3.8.6. sess-key

Использовать отдельный сеансовый ключ. Для шифрования с открытым ключом всегда используется отдельный сеансовый ключ; этот параметр предназначен для шифрования с симметричным ключом, которое по умолчанию использует непосредственно ключ S2K.

Значения: 0, 1
По умолчанию: 0
Применим к: pgp_sym_encrypt

F.34.3.8.7. s2k-mode

Режим алгоритма S2K.

Значения:
0 — Без соли. Опасно!
1 — С солью, но с фиксированным числом итераций.

3 — С переменным числом итераций.
По умолчанию: 3
Применим к: pgp_sym_encrypt

F.34.3.8.8. s2k-count

Число итераций для алгоритма S2K. Оно должно быть не меньше 1024 и не больше 65011712.

По умолчанию: случайное значение между 65536 и 253952
Применим к: pgp_sym_encrypt, только с s2k-mode=3

F.34.3.8.9. s2k-digest-algo

Выбирает алгоритм хеширования, который будет использоваться для вычисления S2K.

Значения: md5, sha1
По умолчанию: sha1
Применим к: pgp_sym_encrypt

F.34.3.8.10. s2k-cipher-algo

Выбирает шифр, который будет использоваться для шифрования отдельного сеансового ключа.

Значения: bf, aes, aes128, aes192, aes256
По умолчанию: используется cipher-algo
Применим к: pgp_sym_encrypt

F.34.3.8.11. unicode-mode

Определяет, преобразовывать ли текстовые данные из внутренней кодировки базы данных в UTF-8 и обратно. Если кодировка базы уже UTF-8, перекодировка не производится, но сообщение помечается как UTF-8. Без данного параметра этого не происходит.

Значения: 0, 1
По умолчанию: 0
Применим к: pgp_sym_encrypt, pgp_pub_encrypt

F.34.3.9. Формирование ключей PGP с применением GnuPG

Формирование нового ключа:

```
gpg --gen-key
```

Предпочитаемый тип ключей: «DSA and Elgamal».

Для шифрования RSA вы должны создать главный ключ либо DSA, либо RSA только для подписания, а затем добавить подключ для шифрования, выполнив команду `gpg --edit-key`.

Просмотр списка ключей:

```
gpg --list-secret-keys
```

Экспорт открытого ключа в формате «ASCII Armor»:

```
gpg -a --export KEYID > public.key
```

Экспорт закрытого ключа в формате «ASCII Armor»:

```
gpg -a --export-secret-keys KEYID > secret.key
```

Прежде чем передавать эти ключи функциям PGP, вы должны применить функцию `dearmor()` к этим ключам. Либо, если вы можете обработать двоичные данные, уберите `-a` из команды.

Дополнительную информацию вы можете получить в руководстве `man gpg`, *The GNU Privacy Handbook* (Руководство GNU по обеспечению конфиденциальности) и другой документации на сайте *http://www.gnupg.org*.

F.34.3.10. Ограничения кода PGP

- Не поддерживается подписывание. Это также означает, что принадлежность подключа шифрования главному ключу не проверяется.
- Не поддерживается использование ключа шифрования в качестве главного ключа. Так как подобная практика обычно не приветствуется, это не должно быть проблемой.
- Нет поддержки нескольких подключей. Это может представляться проблемой, так как такие ключи не редкость. С другой стороны, вы всё равно не должны использовать обычные ключи GPG/PGP с `pgcrypto`, а должны создать новые, учитывая, что это другой сценарий использования.

F.34.4. Низкоуровневые функции шифрования

Эти функции выполняют только шифрование данных; они не предоставляют расширенные возможности шифрования PGP. Таким образом, с ними связаны следующие проблемы:

1. Они используют ключ пользователя непосредственно в качестве ключа шифрования.
2. Они не обеспечивают проверку целостности, которая должна выявлять модификацию зашифрованных данных.
3. Они рассчитаны на то, что пользователи будут управлять всеми параметрами шифрования самостоятельно, даже вектором инициализации.
4. Они не рассчитаны на текст.

Поэтому с появлением поддержки шифрования PGP использовать низкоуровневые функции шифрования не рекомендуется.

```
encrypt(data bytea, key bytea, type text) returns bytea
decrypt(data bytea, key bytea, type text) returns bytea

encrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
decrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
```

Эти функции зашифровывают/расшифровывают данные, применяя метод шифрования, заданный параметром *type*. Строка *type* имеет следующий формат:

алгоритм [- *режим*] [/pad: *дозаполнение*]

где допустимый *алгоритм*:

- `bf` — Blowfish
- `aes` — AES (Rijndael-128, -192 или -256)

допустимый *режим*:

- `cbc` — следующий блок зависит от предыдущего (по умолчанию)
- `ecb` — каждый блок шифруется отдельно (только для тестирования)

и допустимое *дозаполнение*:

- `pkcs` — данные могут быть любой длины (по умолчанию)
- `none` — размер данных должен быть кратен размеру блока шифра

Так что, например, эти вызовы равнозначны:

```
encrypt(data, 'fooz', 'bf')
encrypt(data, 'fooz', 'bf-cbc/pad:pkcs')
```

Для функций `encrypt_iv` и `decrypt_iv` параметр *`iv`* задаёт начальное значение для режима CBC; для ECB он игнорируется. Оно обрезается или дополняется нулями, если его размер не равен ровно размеру блока. В функциях без этого параметра оно по умолчанию заполняется нулями.

F.34.5. Функции получения случайных данных

```
gen_random_bytes(count integer) returns bytea
```

Возвращает криптографически стойкие случайные байты в количестве *`count`*. За один вызов можно получить максимум 1024 байт. Это ограничение предотвращает исчерпание пула энтропии.

```
gen_random_uuid() returns uuid
```

Возвращает UUID версии 4 (случайный).

F.34.6. Замечания

F.34.6.1. Конфигурирование

Модуль `pgcrypto` настраивается согласно установкам, полученным в главном скрипте `configure` Postgres Pro. На его конфигурацию влияют аргументы `--with-zlib` и `--with-openssl`.

При компиляции с zlib шифрующие функции PGP могут сжимать данные перед шифрованием.

При компиляции с OpenSSL будут доступны дополнительные алгоритмы. Кроме того, функции шифрования с открытым ключом будут быстрее, так как OpenSSL содержит оптимизированные функции для работы с большими числами (BIGNUM).

Таблица F.21. Обзор функциональности с и без OpenSSL

Функциональность	**Встроенная**	**С OpenSSL**
MD5	да	да
SHA1	да	да
SHA224/256/384/512	да	да
Другие алгоритмы хеширования	нет	да (Примечание 1)
Blowfish	да	да
AES	да	да
DES/3DES/CAST5	нет	да
Низкоуровневое шифрование	да	да
Симметричное шифрование PGP	да	да
Шифрование PGP с открытым ключом	да	да

Чтобы использовать устаревшие алгоритмы шифрования, такие как DES или Blowfish, при компиляции с OpenSSL версии 3.0.0 и выше, нужно активировать поставщиков этих алгоритмов в файле конфигурации `openssl.cnf`.

Замечания:

1. Автоматически выбирается любой алгоритм хеширования, который поддерживает OpenSSL. Это невозможно с шифрами, они должны поддерживаться явно.

F.34.6.2. Обработка NULL

Как и положено по стандарту SQL, все эти функции возвращают NULL, если один из аргументов — NULL. Это может угрожать безопасности при неаккуратном использовании.

F.34.6.3. Ограничения безопасности

Все функции `pgcrypto` выполняются внутри сервера баз данных. Это означает, что все данные и пароли передаются между функциями `pgcrypto` и клиентскими приложениями открытым текстом. Поэтому вы должны:

1. Подключаться локально или использовать подключения SSL.

2. Доверять и системе, и администратору баз данных.

Если это невозможно, лучше произвести шифрование в клиентском приложении.

Эта реализация не противостоит *атакам по сторонним каналам*. Например, время, требующееся для выполнения функции дешифрования `pgcrypto`, будет разным для разного шифротекста заданного размера.

F.34.6.4. Полезное чтение

- *http://www.gnupg.org/gph/en/manual.html*

  The GNU Privacy Handbook (Руководство GNU по обеспечению конфиденциальности)

- *https://www.openwall.com/crypt/*

  Описывает алгоритм crypt-blowfish.

- *https://www.iusmentis.com/security/passphrasefaq/*

  Как выбрать хороший пароль.

- *http://world.std.com/~reinhold/diceware.html*

  Интересный способ выбора пароля.

- *http://www.interhack.net/people/cmcurtin/snake-oil-faq.html*

  Описывает хорошую и плохую криптографию.

F.34.6.5. Техническая информация

- *https://tools.ietf.org/html/rfc4880*

  Формат сообщений OpenPGP.

- *https://tools.ietf.org/html/rfc1321*

  Алгоритм вычисления дайджеста сообщения MD5.

- *https://tools.ietf.org/html/rfc2104*

  HMAC: Хеширование по ключу для аутентификации сообщений.

- *http://www.usenix.org/events/usenix99/provos.html*

  Сравнение алгоритмов crypt-des, crypt-md5 и bcrypt.

- *http://en.wikipedia.org/wiki/Fortuna_(PRNG)*

  Описание Fortuna CSPRNG.

- *https://jlcooke.ca/random/*

Драйвер /dev/random для Linux на базе Fortuna, написанный Жан-Люком Куком.

F.34.7. Автор

Марко Крин <markokr@gmail.com>

Модуль pgcrypto заимствует код из следующих источников:

Алгоритм	Автор	Источник исходного кода
Шифрование DES	Дэвид Буррен и другие	FreeBSD, libcrypt
Хеширование MD5	Пол-Хеннинг Камп	FreeBSD, libcrypt
Шифрование Blowfish	Solar Designer	www.openwall.com
Шифр Blowfish	Саймон Тэтем	PuTTY
Шифр Rijndael	Брайан Глэдмен	OpenBSD, sys/crypto
Хеш MD5 и SHA1	Проект WIDE	KAME, kame/sys/crypto
SHA256/384/512	Аарон Д. Гиффорд	OpenBSD, sys/crypto
Математика BIGNUM	Майкл Дж. Фромбергер	dartmouth.edu/~sting/sw/imath

F.35. pg_freespacemap

Модуль pg_freespacemap предоставляет средства для исследования карты свободного пространства (FSM). В нём реализована функция pg_freespace, точнее, две перегруженных функции. Эти функции показывают значение, записанное в карте свободного пространства для данной страницы, либо для всех страниц отношения.

По умолчанию его использование разрешено только суперпользователям и членам роли pg_stat_scan_tables. Дать доступ другим можно с помощью GRANT.

F.35.1. Функции

pg_freespace(rel regclass IN, blkno bigint IN) returns int2

Возвращает объём свободного пространства на странице для отношения, заданного параметром blkno, согласно FSM.

pg_freespace(rel regclass IN, blkno OUT bigint, avail OUT int2)

Выдаёт объём свободного пространства на каждой странице отношения, согласно FSM. Возвращается набор кортежей (blkno bigint, avail int2), по одному кортежу для каждой страницы в отношении.

Значения, хранимые в карте свободного пространства, не являются точными. Они округляются до 1/256 величины BLCKSZ (до 32 байт со значением BLCKSZ по умолчанию) и не поддерживаются в актуальном состоянии при каждом добавлении и изменении кортежей.

Для индексов отслеживаются полностью неиспользованные страницы, а не свободное пространство в страницах. Таким образом, эти значения отражают только то, что страница занята или свободна.

> **Примечание**
>
> Интерфейс был изменён в версии 8.4, в соответствии с нововведениями реализации FSM, которые появились в этой версии.

F.35.2. Пример вывода

```
postgres=# SELECT * FROM pg_freespace('foo');
 blkno | avail
-------+-------
     0 |     0
     1 |     0
     2 |     0
     3 |    32
     4 |   704
     5 |   704
     6 |   704
     7 |  1216
     8 |   704
     9 |   704
    10 |   704
    11 |   704
    12 |   704
    13 |   704
    14 |   704
    15 |   704
    16 |   704
    17 |   704
    18 |   704
    19 |  3648
(20 rows)

postgres=# SELECT * FROM pg_freespace('foo', 7);
 pg_freespace
--------------
         1216
(1 row)
```

F.35.3. Автор

Исходную версию разработал Марк Кирквуд <markir@paradise.net.nz>. Для версии 8.4 с новой реализацией FSM код адаптировал Хейкки Линнакангас <heikki@enterprisedb.com>

F.36. pg_pathman

pg_pathman — это расширение Postgres Pro, реализующее оптимизированное решение для секционирования больших и распределённых баз данных. Используя pg_pathman, вы можете:

- Секционировать большие базы данных, не прерывая их работу.
- Ускорять выполнение запросов с секционированными таблицами.
- Управлять существующими и добавлять новые секции на лету.
- Добавлять в качестве секций сторонние таблицы.
- Соединять секционированные таблицы для операций чтения и записи.

Это расширение совместимо с Postgres Pro 9.5 и новее.

F.36.1. Установка и подготовка

Расширение pg_pathman включено в состав Postgres Pro. Установив Postgres Pro, выполните следующие действия, чтобы подготовить pg_pathman к работе:

1. Добавьте pg_pathman в переменную shared_preload_libraries в файле postgresql.conf:

```
shared_preload_libraries = 'pg_pathman'
```

> **Важно**
>
> pg_pathman может конфликтовать с другими расширениями, использующими те же функции для перехвата управления. Например, возможен конфликт pg_pathman с pg_stat_statements, так как оба эти расширения используют функцию ProcessUtility_hook. Во избежание подобных проблем pg_pathman должен быть всегда последним в списке библиотек: shared_preload_libraries = 'pg_stat_statements, pg_pathman'

2. Перезапустите Postgres Pro, чтобы изменения вступили в силу.
3. Создайте расширение pg_pathman следующим образом:

```
CREATE SCHEMA pathman;
GRANT USAGE ON SCHEMA pathman TO PUBLIC;
CREATE EXTENSION pg_pathman WITH SCHEMA pathman;
```

> **Важно**
>
> Чтобы ваши обращения к функциям pg_pathman были защищены от атак с подменой search_path (см. CREATE EXTENSION), устанавливайте это расширение только в чистую схему, где никто, кроме суперпользователей, не имеет права CREATE для создания объектов базы данных.

После того как расширение pg_pathman будет создано, вы можете приступить к секционированию таблиц.

> **Примечание**
>
> Во время установки pg_pathman создаёт несколько политик RLS для ограничения доступа к собственным таблицам. Однако ядро Postgres Pro не поддерживает в полной мере выгрузку/восстановление дампов с расширениями, использующими в своих скриптах операторы CREATE POLICY. В связи с этим при восстановлении дампа базы, в которой установлено расширение pg_pathman, вы получите сообщения об ошибках вида:
>
> ```
> ERROR: policy "allow_select" for table "pathman_config" already exists
> ```
>
> (ОШИБКА: политика "allow_select" для таблицы "pathman_config" уже существует) Их следует игнорировать, так как на полноту восстанавливаемых данных эти ошибки не влияют.

> **Подсказка**
>
> Вы также можете скомпилировать pg_pathman из исходного кода, выполнив следующую команду в каталоге pg_pathman:
>
> ```
> make install USE_PGXS=1
> ```
>
> Завершив эту операцию, выполните следующие действия для окончания установки.
>
> Также не забудьте дополнительно установить переменную PG_CONFIG, если вы хотите испытать pg_pathman в нестандартной сборке Postgres Pro. Подробнее об этом вы можете прочитать *здесь*.

Включать/отключать pg_pathman или его определённые узлы можно с помощью переменных GUC. За подробностями обратитесь к Подразделу F.36.5.1.

Если вы хотите полностью отключить pg_pathman для ранее секционированной таблицы, воспользуйтесь функцией disable_pathman_for():

```
SELECT disable_pathman_for('range_rel');
```

Все секции и данные останутся неизменными и будут обрабатываться стандартным механизмом наследования Postgres Pro.

F.36.1.1. Обновление расширения pg_pathman

Если у вас уже была установлена предыдущая версия pg_pathman, выполните следующие действия для установки новой версии:

1. Установите Postgres Pro.
2. Перезапустите кластер Postgres Pro.
3. Если у вас уже была установлена предыдущая основная версия pg_pathman (в её номере другая вторая цифра), выполните следующие действия для установки новой версии:

   ```
   ALTER EXTENSION pg_pathman UPDATE TO версия;
   SET pg_pathman.enable = t;
   ```

   Здесь *версия* — это номер основной версии pg_pathman, например, 1.5.

   Узнать текущую версию pg_pathman можно, воспользовавшись функцией pathman_version().

F.36.2. Использование

Выбор стратегии секционирования

Осуществление неблокирующего переноса данных

Секционирование по одному выражению

Секционирование по составному ключу

Реализация многоуровневого секционирования

Управление секциями

По мере увеличения базы данных механизмы индексирования могут становиться неэффективными и запросы могут выполняться намного медленнее. Для повышения производительности, улучшения масштабируемости и оптимизации процессов администрирования вы можете использовать секционирование — разделение большой таблицы на множество меньших по размеру, когда все строки размещаются в секциях согласно ключу разбиения.

Исторически Postgres Pro поддерживал секционирование через механизм наследования, когда каждая секция создавалась в виде дочерней таблицы с ограничением-проверкой. В Postgres Pro 10 появилась поддержка декларативного секционирования, которая также полагается на наследование. При таком подходе планировщик запросов должен выполнить полный перебор и проверку условий ограничений для каждой секции, чтобы построить план выполнения запроса, что влечёт замедление запросов к таблицам с большим количеством секций. Расширение pg_pathman использует оптимизированные алгоритмы планирования и функции секционирования, учитывающие внутреннюю структуру секционированных таблиц, что позволяет добиться лучшей производительности. Более подробно детали реализации pg_pathman описаны в Подразделе F.36.4.

F.36.2.1. Выбор стратегии секционирования

Расширение pg_pathman поддерживает следующие стратегии секционирования:

- По хешу — строки сопоставляются с секциями с использованием универсальной функции хеширования. Выберите эту стратегию, если в большинстве запросов будет выполняться поиск точного соответствия.
- По диапазонам — строки сопоставляются с секциями по диапазонам разбивающего ключа, назначаемым каждой секции. Выберите эту стратегию, если ваша база данных содержит числовые данные, которые, скорее всего, будут представлять интерес как значения в интервалах. Например, вас могут интересовать исторические данные по годам или результаты экспериментов в определённых числовых диапазонах. Для получения выигрыша в производительности `pg_pathman` использует алгоритм бинарного поиска.

По умолчанию `pg_pathman` переносит все данные из родительской таблицы в создаваемые секции сразу (производится *блокирующее секционирование*). При таком подходе вы можете изменить структуру таблицы в одной транзакции, но если объём данных велик, это может привести к приостановке работы. Если важно, чтобы работа не прерывалась, вы можете выполнить *параллельное секционирование*. В этом случае `pg_pathman` записывает все новые данные в созданные секции, но сохраняет исходные данные в родительской таблице, пока вы явно не перенесёте их. Это позволяет секционировать большие базы данных, не прерывая работу, так как вы можете выбрать удобное время для переноса данных и переносить их небольшими порциями, не блокируя другие транзакции. Подробнее параллельное секционирование описано в Подразделе F.36.2.2.

F.36.2.1.1. Организация секционирования по хешу

Чтобы выполнить секционирование по хешу с применением `pg_pathman`, воспользуйтесь функцией `create_hash_partitions()`:

```
create_hash_partitions(parent_relid     REGCLASS,
                       expression       TEXT,
                       partitions_count INTEGER,
                       partition_data   BOOLEAN DEFAULT TRUE,
                       partition_names  TEXT[] DEFAULT NULL,
                       tablespaces      TEXT[] DEFAULT NULL)
```

Модуль `pg_pathman` создаёт указанное число секций, используя хеш-функцию. Вы можете также указать имена секций и табличных пространств, задав параметры `partition_names` и `tablespaces`, соответственно.

После того как таблица разделена на секции, удалять или добавлять секции в ней нельзя. Однако при необходимости определённую секцию можно заменить другой таблицей:

```
replace_hash_partition(old_partition       REGCLASS,
                       new_partition       REGCLASS,
                       lock_parent         BOOL DEFAULT TRUE);
```

Если параметр `lock_parent` равен `true`, никакие запросы `INSERT`/`UPDATE`/`ALTER TABLE` в родительской таблице не разрешаются.

Если вы опустите необязательный параметр `partition_data` или зададите для него значение `true`, все данные из родительской таблицы будут перенесены в секции. Модуль `pg_pathman` заблокирует эту таблицу для других транзакций до завершения переноса данных. Чтобы избежать приостановки работы, вы можете передать в параметре `partition_data` значение `false` и затем вызвать функцию `partition_table_concurrently()` для переноса данных без блокирования других запросов. За подробностями обратитесь к Подразделу F.36.2.2.

F.36.2.1.2. Организация секционирования по диапазонам

Модуль `pg_pathman` предоставляет функцию `create_range_partitions()` для разбиения по диапазонам. Эта функция создаёт секции, исходя из заданного интервала и начального значения ключа разбиения. Новые секции будут создаваться автоматически при добавлении данных, не попадающих в ранее охваченный интервал.

```
create_range_partitions(parent_relid   REGCLASS,
                        expression     TEXT,
                        start_value    ANYELEMENT,
                        p_interval     ANYELEMENT | INTERVAL,
                        p_count        INTEGER DEFAULT NULL,
                        partition_data BOOLEAN DEFAULT TRUE)
```

Модуль `pg_pathman` создаёт секции в зависимости от переданных параметров. Если необязательный параметр `p_count` опущен, `pg_pathman` вычисляет требуемое число секций, исходя из заданного начального значения и интервала. Если затем будут добавляться данные вне ранее заданного диапазона, `pg_pathman` создаст новые секции автоматически, сохраняя указанный интервал. При таком подходе все секции оказываются одного размера, что может способствовать ускорению запросов и упрощению управления данными.

Также вы можете задать массив, определяющий границы создаваемых секций, в параметре `bounds`:

```
create_range_partitions(parent_relid    REGCLASS,
                        expression      TEXT,
                        bounds          ANYARRAY,
                        partition_names TEXT[] DEFAULT NULL,
                        tablespaces     TEXT[] DEFAULT NULL,
                        partition_data  BOOLEAN DEFAULT TRUE)
```

Если требуется, вы можете также использовать функции управления секциями и добавлять секции вручную. Например, если между созданными секциями образовался промежуток, `pg_pathman` не сможет заполнить его новой секцией в автоматическом режиме.

По умолчанию все данные из родительской таблицы будут перенесены в указанное количество секций. Модуль `pg_pathman` заблокирует эту таблицу для других транзакций до завершения переноса данных. Чтобы избежать приостановки работы, вы можете передать в параметре `partition_data` значение `false` и затем вызвать функцию `partition_table_concurrently()` для переноса данных без блокирования других запросов. За подробностями обратитесь к Подразделу F.36.2.2.

F.36.2.2. Осуществление неблокирующего переноса данных

Если важно не допустить прерывания работы, вы можете произвести секционирование в параллельном режиме, установив для параметра `partition_data` значение `false`. В этом случае `pg_pathman` создаст пустые секции и оставит все исходные данные в родительской таблице. При этом все новые записи будут попадать в созданные секции. Позднее вы сможете переместить все начальные данные в соответствующие секции, не блокируя другие запросы, воспользовавшись функцией `partition_table_concurrently()`:

```
partition_table_concurrently(relation   REGCLASS,
                             batch_size INTEGER DEFAULT 1000,
                             sleep_time FLOAT8 DEFAULT 1.0)
```

Здесь:

- `relation` — родительская таблица.
- `batch_size` — количество строк, которое должно копироваться из родительской таблицы в секции за один раз. Этот параметр может принимать любое целое значение от 1 до 10000.
- `sleep_time` — интервал времени между попытками переноса данных, в секундах.

Модуль `pg_pathman` запускает фоновый рабочий процесс для переноса данных из родительской таблицы в секции маленькими порциями, размер которых задаётся параметром `batch_size`. Если одна или несколько строк в порции оказались заблокированы другими запросами, `pg_pathman` ждёт заданное время (`sleep_time`) и повторяет попытку (до 60 раз). За процессом переноса данных

можно наблюдать в представлении pathman_concurrent_part_tasks, показывающем количество строк, обработанных на данный момент:

```
[user]postgres: select * from pathman_concurrent_part_tasks ;
 userid |  pid  | dbid  | relid | processed | status
--------+-------+-------+-------+-----------+---------
 user   | 20012 | 12413 | test  |    334000 | working
(1 row)
```

Если потребуется остановить перенос данных, вы можете в любое время выполнить функцию stop_concurrent_part_task():

```
SELECT stop_concurrent_part_task(relation REGCLASS);
```

pg_pathman завершит перенос текущей порции и прекратит процесс переноса.

> **Подсказка**
>
> Когда pg_pathman перенесёт все данные из родительской таблицы, вы можете исключить её из плана запроса. За подробностями обратитесь к описанию функции set_enable_parent().

F.36.2.3. Секционирование по одному выражению

Для стратегий секционирования по диапазону и хешу pg_pathman поддерживает разбиение по выражению, возвращающему одно скалярное значение. Выражением разбиения может быть ссылка на столбец таблицы или вычисление ключа разбиения с использованием значений одного или нескольких столбцов.

> **Подсказка**
>
> Если вас интересует разбиение таблицы по значению кортежа, обратитесь к Подразделу F.36.2.4.

Для разбиения таблицы по выражению используйте функции секционирования pg_pathman. Выражение разбиения должно соответствовать следующим условиям:

- В выражении должен фигурировать минимум один столбец секционируемой таблицы.
- Все фигурирующие в нём столбцы должны иметь свойство NOT NULL.
- В выражении нельзя обращаться к системным атрибутам, таким как oid, xmin, xmax и т. д.
- В выражение нельзя включать подзапросы.
- Все функции, используемые выражением, должны быть помечены как IMMUTABLE.

Так как выражение может возвращать значение практически любого типа, возвращаемое значение нужно привести к типу, для которого выполняется разбиение.

Для обращения к секции необходимо использовать в точности то выражение, по которому выполнено разбиение. В противном случае pg_pathman не сможет оптимизировать запрос. Просмотреть выражения разбиения для всех секционированных таблиц можно в таблице pathman_config.

F.36.2.3.1. Примеры

Предположим, что у вас есть таблица test, содержащая некоторые данные jsonb:

```
CREATE TABLE test(col jsonb NOT NULL);
INSERT INTO test
```

```
SELECT format('{"key": %s, "date": "%s", "value": "%s"}',
              i, current_date, md5(i::text))::jsonb
FROM generate_series(1, 10000 * 10) as g(i);
```

Для разбиения этой таблицы по диапазонам значения `key`, вам нужно извлечь это значение из объекта `jsonb` и преобразовать его в числовой тип, например, в `bigint`:

```
SELECT create_range_partitions('test', '(col->>''key'')::bigint', 1, 10000, 10);
```

В результате `pg_pathman` разделит родительскую таблицу на десять секций и поместит в каждую 10000 строк:

```
SELECT * FROM pathman_partition_list;
 parent | partition | parttype |              expr               | range_min |
 range_max
--------+-----------+----------+---------------------------------+-----------
+-----------
 test   | test_1    |        2 | ((col ->> 'key'::text))::bigint | 1         | 10001
 test   | test_2    |        2 | ((col ->> 'key'::text))::bigint | 10001     | 20001
 test   | test_3    |        2 | ((col ->> 'key'::text))::bigint | 20001     | 30001
 test   | test_4    |        2 | ((col ->> 'key'::text))::bigint | 30001     | 40001
 test   | test_5    |        2 | ((col ->> 'key'::text))::bigint | 40001     | 50001
 test   | test_6    |        2 | ((col ->> 'key'::text))::bigint | 50001     | 60001
 test   | test_7    |        2 | ((col ->> 'key'::text))::bigint | 60001     | 70001
 test   | test_8    |        2 | ((col ->> 'key'::text))::bigint | 70001     | 80001
 test   | test_9    |        2 | ((col ->> 'key'::text))::bigint | 80001     | 90001
 test   | test_10   |        2 | ((col ->> 'key'::text))::bigint | 90001     | 100001
(10 rows)
```

F.36.2.4. Секционирование по составному ключу

Используя `pg_pathman`, вы также можете реализовать диапазонное секционирование по составному ключу. Составной ключ образуется из двух или нескольких разделённых запятыми элементов, которыми могут быть ссылки на столбцы или выражения, извлекающие значения из таблицы. Выражения, определяющие составной ключ, должны удовлетворять условиям, перечисленным в Подразделе F.36.2.3.

Хотя `pg_pathman` не поддерживает автоматическое создание секций с составным ключом, вы можете добавлять секции, используя функцию `add_range_partition()`. Обычно это происходит так:

1. Включите автоматическое наименование секций для вашей таблицы, вызвав функцию `create_naming_sequence()`.
2. Создайте составной ключ разбиения.
3. Зарегистрируйте таблицу, которую вы будете секционировать с помощью `pg_pathman`, воспользовавшись функцией `add_to_pathman_config()`.
4. Добавьте секции на основе составного ключа разбиения, вызвав функцию `add_range_partition()`.

F.36.2.4.1. Примеры

Предположим, что у вас есть таблица `test`, содержащая некоторые данные с датами:

```
CREATE TABLE test (logdate date NOT NULL, comment text);
```

Для разбиения таких данных по месяцам и годам вам нужно создать составной ключ:

```
CREATE TYPE test_key AS (year float8, month float8);
```

Для включения автоматического именования секций выполните функцию `create_naming_sequence()`, передав в качестве аргумента имя таблицы:

```
SELECT create_naming_sequence('test');
```

Зарегистрируйте таблицу `test` в `pg_pathman`, указав ключ разбиения, который вы намерены использовать:

```
SELECT add_to_pathman_config('test',
                             '( extract(year from logdate),
                                extract(month from logdate) )::test_key',
                            NULL);
```

Создайте секцию, включающую все данные в интервале десяти лет, начиная с января текущего кода:

```
SELECT add_range_partition('test',
                           (extract(year from current_date), 1)::test_key,
                           (extract(year from current_date + '10 years'::interval),
 1)::test_key);
```

F.36.2.5. Реализация многоуровневого секционирования

`pg_pathman` поддерживает многоуровневое секционирование для стратегий разбиения и по хешу, и по диапазонам. Вы можете использовать эти стратегии в любом сочетании: таблица, секционирования по хешам или диапазонам, затем может быть секционирована и по диапазонам, и по хешу.

Чтобы разбить существующую секцию на несколько дочерних, используйте обычные функции секционирования `pg_pathman`, как описано в Подразделе F.36.2.1, передавая имя данной секции в параметре *`parent_relid`*. Точные имена секций вы можете узнать из представления pathman_partition_list.

При реализации диапазонного разбиения с внутренним диапазонным разбиением можно выбрать для внутреннего разбиения либо другое выражение, либо то же, что задано для родительской таблицы. В первом случае, если выбранный диапазон больше диапазона родительской таблицы, будут использоваться только те дочерние секции, диапазон которых пересекается с родительским. Другие дочерние секции останутся пустыми, если только не произойдёт объединение их родительской секции с соседними, в результате которого хотя бы частично будет охвачен их диапазон.

F.36.2.5.1. Примеры

Предположим, что у вас есть таблица `journal`, секционированная по месяцам:

```
-- создание пустой таблицы
CREATE TABLE journal (
id      SERIAL,
dt      TIMESTAMP NOT NULL,
level   INTEGER,
msg     TEXT);

-- добавление в таблицу некоторых данных
INSERT INTO journal (dt, level, msg)
SELECT g, random() * 6, md5(g::text)
FROM generate_series('2015-01-01'::date, '2015-12-31'::date, '1 minute') as g;

-- секционирование таблицы по диапазонам
SELECT create_range_partitions('journal', 'dt', '2015-01-01'::date, '1
 month'::interval);
```

Если в какой-то момент становится выгоднее иметь меньшие секции, вы можете дополнительно разбить существующие по диапазонам или по секциям. Например, чтобы разбить секцию `journal_1` на меньшие по дням, выполните:

```
SELECT create_range_partitions('journal_1', 'dt', '2015-01-01'::date, '1
 day'::interval);
```

Подобным образом вы можете создать вложенные секции с разбиением по хешу. Например, так можно разбить секцию `journal_2` на пять секций, используя столбец `id` в качестве ключа разбиения:

```
SELECT create_hash_partitions('journal_2', 'id', '5');
```

F.36.2.6. Управление секциями

`pg_pathman` предоставляет набор функций для простого управления секциями. За подробностями обратитесь к Подразделу F.36.5.3.4.

F.36.3. Примеры

F.36.3.1. Общие рекомендации

- Так можно получить столбец `partition`, содержащий имена нижележащих секций, воспользовавшись системным атрибутом `tableoid`:

  ```
  SELECT tableoid::regclass AS partition, * FROM partitioned_table;
  ```

- Несмотря на то, что индексы в родительской таблице не очень полезны (так как предполагается, что она пуста), они выполняют роль прототипов для создания индексов в секциях. Для каждого индекса в родительской таблице `pg_pathman` создаёт подобный индекс в каждой секции.

- Получить список всех текущих задач параллельного разбиения можно в представлении `pathman_concurrent_part_tasks`:

  ```
  SELECT * FROM pathman_concurrent_part_tasks;
  userid  | pid  | dbid  | relid | processed | status
  --------+------+-------+-------+-----------+---------
  user    | 7367 | 16384 | test  |    472000 | working
  (1 row)
  ```

- Представление `pathman_partition_list` в сочетании с `drop_range_partition()` может использоваться для удаления диапазонных секций более гибким образом по сравнению с обычным `DROP TABLE`:

  ```
  SELECT drop_range_partition(partition, false) /* перенос данных в родительскую
   таблицу */
  FROM pathman_partition_list
  WHERE parent = 'part_test'::regclass AND range_min::int < 500;
  NOTICE:  1 rows copied from part_test_11
  NOTICE:  100 rows copied from part_test_1
  NOTICE:  100 rows copied from part_test_2
  drop_range_partition
  ----------------------
  dummy_test_11
  dummy_test_1
  dummy_test_2
  (3 rows)
  ```

- Вы можете сделать сторонние таблицы секциями с помощью функции `attach_range_partition()`. В результате строки, добавляемые в родительскую таблицу, будут

перенаправляться в эти сторонние секции механизмом `PartitionFilter`. По умолчанию добавление строк в такие секции разрешается только при использовании обёртки сторонних данных `postgres_fdw`. Это поведение управляется переменной `pg_pathman.insert_into_fdw`. Для её изменения необходимо иметь права суперпользователя.

F.36.3.2. Секционирование по хешу

Рассмотрим пример секционирования таблицы по хешу. Для начала создадим таблицу с целочисленным столбцом:

```
CREATE TABLE items (
id       SERIAL PRIMARY KEY,
name     TEXT,
code     BIGINT);

INSERT INTO items (id, name, code)
SELECT g, md5(g::text), random() * 100000
FROM generate_series(1, 100000) as g;
```

Теперь выполним функцию `create_hash_partitions()` с подходящими аргументами:

```
SELECT create_hash_partitions('items', 'id', 100);
```

Этот запрос создаст новые секции и переместит в них данные из родительской таблицы.

Пример запроса, выполняющего фильтрацию по ключу разбиения:

```
SELECT * FROM items WHERE id = 1234;
  id  |               name               | code
------+----------------------------------+------
 1234 | 81dc9bdb52d04dc20036dbd8313ed055 | 1855
(1 row)

EXPLAIN SELECT * FROM items WHERE id = 1234;
QUERY PLAN
-------------------------------------------------------------------------------------
Append  (cost=0.28..8.29 rows=0 width=0)
->  Index Scan using items_34_pkey on items_34  (cost=0.28..8.29 rows=0 width=0)
Index Cond: (id = 1234)
```

Заметьте, узел `Append` содержит только одно дочернее сканирование, соответствующее предложению `WHERE`.

> **Важно**
>
> Обратите внимание на тот факт, что `pg_pathman` исключает родительскую таблицу из плана запроса.

Чтобы обратиться к родительской таблице, используйте модификатор `ONLY`:

```
EXPLAIN SELECT * FROM ONLY items;
QUERY PLAN
-----------------------------------------------------
Seq Scan on items  (cost=0.00..0.00 rows=1 width=45)
```

F.36.3.3. Секционирование по диапазонам

Рассмотрим пример секционирования по диапазонам. Давайте создадим таблицу, содержащую сообщения журнала:

```
CREATE TABLE journal (
id      SERIAL,
dt      TIMESTAMP NOT NULL,
level   INTEGER,
msg     TEXT);

-- подобный индекс будет также создан для каждой секции
CREATE INDEX ON journal(dt);

-- генерируются некоторые данные
INSERT INTO journal (dt, level, msg)
SELECT g, random() * 6, md5(g::text)
FROM generate_series('2015-01-01'::date, '2015-12-31'::date, '1 minute') as g;
```

Выполним функцию `create_range_partitions()`, чтобы создать секции, которые будут содержать данные за один день:

```
SELECT create_range_partitions('journal', 'dt', '2015-01-01'::date, '1 day'::interval);
```

Этот запрос создаст 364 секции и переместит в них данные из родительской таблицы.

Новые секции добавляются автоматически триггером INSERT, но это можно сделать и вручную, с помощью следующих функций:

```
-- добавление новой секции с заданным диапазоном
SELECT add_range_partition('journal', '2016-01-01'::date, '2016-01-07'::date);

-- добавление новой секции с диапазоном по умолчанию
SELECT append_range_partition('journal');
```

Первая создаёт новую секцию с заданным диапазоном. Вторая создаёт новую секцию с диапазоном, заданным при первоначальном разбиении, и добавляет её в конец списка секций. Также можно присоединить существующую таблицу в качестве секции. Например, мы можем подключить таблицу архива (это может быть даже сторонняя таблица с другого сервера) с историческими данными:

```
CREATE FOREIGN TABLE journal_archive (
id      INTEGER NOT NULL,
dt      TIMESTAMP NOT NULL,
level   INTEGER,
msg     TEXT)
SERVER archive_server;

SELECT attach_range_partition('journal', 'journal_archive', '2014-01-01'::date,
 '2015-01-01'::date);
```

> **Важно**
>
> Присоединённая таблица должна содержать такие же столбцы, что и секционируемая, за исключением удалённых. Эти столбцы должны иметь те же типы, правила сортировки и характеристики `NOT NULL`, что и исходные.

Для слияния двух соседних секций используйте функцию `merge_range_partitions()`:

```
SELECT merge_range_partitions('journal_archive', 'journal_1');
```

Чтобы разделить секцию по значению, воспользуйтесь функцией `split_range_partition()`:

```
SELECT split_range_partition('journal_366', '2016-01-03'::date);
```

Чтобы отсоединить секцию, воспользуйтесь функцией `detach_range_partition()`:

```
SELECT detach_range_partition('journal_archive');
```

Пример запроса, выполняющего фильтрацию по ключу разбиения:

```
SELECT * FROM journal WHERE dt >= '2015-06-01' AND dt < '2015-06-03';
id      |         dt          | level |               msg
--------+---------------------+-------+----------------------------------
217441  | 2015-06-01 00:00:00 |     2 | 15053892d993ce19f580a128f87e3dbf
217442  | 2015-06-01 00:01:00 |     1 | 3a7c46f18a952d62ce5418ac2056010c
217443  | 2015-06-01 00:02:00 |     0 | 92c8de8f82faf0b139a3d99f2792311d
...
(2880 rows)

EXPLAIN SELECT * FROM journal WHERE dt >= '2015-06-01' AND dt < '2015-06-03';
QUERY PLAN
------------------------------------------------------------------
Append  (cost=0.00..58.80 rows=0 width=0)
->  Seq Scan on journal_152  (cost=0.00..29.40 rows=0 width=0)
->  Seq Scan on journal_153  (cost=0.00..29.40 rows=0 width=0)
(3 rows)
```

F.36.4. Внутреннее устройство

Расширение `pg_pathman` сохраняет конфигурацию разбиения в таблице `pathman_config`; каждая её строка содержит запись для одной секционированной таблицы (название отношения, столбец разбиения и тип секционирования). На этапе инициализации модуль `pg_pathman` кеширует некоторую информацию дочерних секций в общей памяти, а затем она может использоваться при построении плана. Когда начинает выполняться запрос `SELECT`, `pg_pathman` проходит по дереву условий в поиске выражений вида:

```
VARIABLE OP CONST
```

где `VARIABLE` — это ключ разбиения, `OP` — оператор сравнения (поддерживаются =, <, <=, >, >=), `CONST` — скалярное значение. Например:

```
WHERE id = 150
```

Затем, учитывая стратегию секционирования и оператор условия, `pg_pathman` ищет соответствующие секции и строит план.

F.36.4.1. Дополнительные узлы плана

`pg_pathman` предоставляет несколько *нестандартных узлов плана*, позволяющих сократить время выполнения, а именно:

- `RuntimeAppend` (переопределяет узел плана `Append`)
- `RuntimeMergeAppend` (переопределяет узел плана `MergeAppend`)
- `PartitionFilter` (выполняет роль триггеров INSERT)
- `PartitionRouter` реализует межсекционные UPDATE вместо триггеров

`PartitionFilter` действует как *прокси-узел* для дочерних узлов INSERT, то есть он может перенаправлять выходные кортежи в соответствующие секции:

```
EXPLAIN (COSTS OFF)
INSERT INTO partitioned_table
SELECT generate_series(1, 10), random();
               QUERY PLAN
----------------------------------------
 Insert on partitioned_table
```

```
   ->  Custom Scan (PartitionFilter)
         ->  Subquery Scan on "*SELECT*"
               ->  Result
(4 rows)
```

Узел `PartitionRouter` представляет собой ещё один промежуточный узел, используемый вместе с `PartitionFilter` для выполнения межсекционных операций `UPDATE`, имеющих место, например, при изменении одного из столбцов ключа разбиения.

> **Важно**
>
> Узел `PartitionRouter` преобразует межсекционные команды `UPDATE` в `DELETE + INSERT`. В Postgres Pro до 11 версии эта операция небезопасна, так как `pg_pathman` не имеет возможности определить, была ли изменённая строка удалена или перемещена в другую секцию.

По умолчанию узел `PartitionRouter` отключён во избежание нежелательных побочных эффектов. Чтобы включить его, задайте для параметра `pg_pathman.enable_partitionrouter` значение `on`.

```
EXPLAIN (COSTS OFF)
UPDATE partitioned_table
SET value = value + 1 WHERE value = 2;
                    QUERY PLAN
--------------------------------------------------
 Update on partitioned_table_0
   ->  Custom Scan (PartitionRouter)
         ->  Custom Scan (PartitionFilter)
               ->  Seq Scan on partitioned_table_0
                     Filter: (value = 2)
(5 rows)
```

`RuntimeAppend` и `RuntimeMergeAppend` имеют много общего: они оказываются полезными, когда условие WHERE принимает вид:

```
VARIABLE OP PARAM
```

Подобные выражения уже не могут быть оптимизированы во время планирования, так как значение параметра оказывается неизвестным до стадии выполнения. Решить эту проблему можно, включив *процедуру анализа условия WHERE* в первоначальный код узла `Append`, таким образом позволяя ему выбирать только нужные варианты сканирования из всего набора планируемых сканирований. Это по сути сводится к созданию нестандартного узла, способного выполнять такие проверки.

Есть по меньше мере несколько ситуаций, которые демонстрируют полезность таких узлов:

```
/* создать таблицу, которую мы будем секционировать */
CREATE TABLE partitioned_table(id INT NOT NULL, payload REAL);

/* вставить произвольные данные */
INSERT INTO partitioned_table
SELECT generate_series(1, 1000), random();

/* выполнить секционирование */
SELECT create_hash_partitions('partitioned_table', 'id', 100);

/* создать обычную таблицу */
CREATE TABLE some_table AS SELECT generate_series(1, 100) AS VAL;
```

- id = (select ... limit 1)

```
EXPLAIN (COSTS OFF, ANALYZE) SELECT * FROM partitioned_table
WHERE id = (SELECT * FROM some_table LIMIT 1);
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------
 Custom Scan (RuntimeAppend) (actual time=0.030..0.033 rows=1 loops=1)
   InitPlan 1 (returns $0)
     ->  Limit (actual time=0.011..0.011 rows=1 loops=1)
           ->  Seq Scan on some_table (actual time=0.010..0.010 rows=1 loops=1)
   ->  Seq Scan on partitioned_table_70 partitioned_table (actual time=0.004..0.006
 rows=1 loops=1)
         Filter: (id = $0)
         Rows Removed by Filter: 9
 Planning time: 1.131 ms
 Execution time: 0.075 ms
(9 rows)

/* отключить узел RuntimeAppend */
SET pg_pathman.enable_runtimeappend = f;

EXPLAIN (COSTS OFF, ANALYZE) SELECT * FROM partitioned_table
WHERE id = (SELECT * FROM some_table LIMIT 1);
                                      QUERY PLAN
-------------------------------------------------------------------------------------
 Append (actual time=0.196..0.274 rows=1 loops=1)
   InitPlan 1 (returns $0)
     ->  Limit (actual time=0.005..0.005 rows=1 loops=1)
           ->  Seq Scan on some_table (actual time=0.003..0.003 rows=1 loops=1)
   ->  Seq Scan on partitioned_table_0 (actual time=0.014..0.014 rows=0 loops=1)
         Filter: (id = $0)
         Rows Removed by Filter: 6
   ->  Seq Scan on partitioned_table_1 (actual time=0.003..0.003 rows=0 loops=1)
         Filter: (id = $0)
         Rows Removed by Filter: 5
         ... /* more plans follow */
 Planning time: 1.140 ms
 Execution time: 0.855 ms
(306 rows)
```

- id = ANY (select ...)

```
EXPLAIN (COSTS OFF, ANALYZE) SELECT * FROM partitioned_table
WHERE id = any (SELECT * FROM some_table limit 4);
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------
 Nested Loop (actual time=0.025..0.060 rows=4 loops=1)
   ->  Limit (actual time=0.009..0.011 rows=4 loops=1)
         ->  Seq Scan on some_table (actual time=0.008..0.010 rows=4 loops=1)
   ->  Custom Scan (RuntimeAppend) (actual time=0.002..0.004 rows=1 loops=4)
         ->  Seq Scan on partitioned_table_70 partitioned_table (actual
 time=0.001..0.001 rows=10 loops=1)
         ->  Seq Scan on partitioned_table_26 partitioned_table (actual
 time=0.002..0.003 rows=9 loops=1)
         ->  Seq Scan on partitioned_table_27 partitioned_table (actual
 time=0.001..0.002 rows=20 loops=1)
         ->  Seq Scan on partitioned_table_63 partitioned_table (actual
 time=0.001..0.002 rows=9 loops=1)
 Planning time: 0.771 ms
```

```
 Execution time: 0.101 ms
(10 rows)

/* отключить узел RuntimeAppend */
SET pg_pathman.enable_runtimeappend = f;

EXPLAIN (COSTS OFF, ANALYZE) SELECT * FROM partitioned_table
WHERE id = any (SELECT * FROM some_table limit 4);
                                        QUERY PLAN
-----------------------------------------------------------------------------------------------
 Nested Loop Semi Join (actual time=0.531..1.526 rows=4 loops=1)
   Join Filter: (partitioned_table.id = some_table.val)
   Rows Removed by Join Filter: 3990
   ->  Append (actual time=0.190..0.470 rows=1000 loops=1)
         ->  Seq Scan on partitioned_table (actual time=0.187..0.187 rows=0 loops=1)
         ->  Seq Scan on partitioned_table_0 (actual time=0.002..0.004 rows=6
 loops=1)
         ->  Seq Scan on partitioned_table_1 (actual time=0.001..0.001 rows=5
 loops=1)
         ->  Seq Scan on partitioned_table_2 (actual time=0.002..0.004 rows=14
 loops=1)
... /* 96 scans follow */
   ->  Materialize (actual time=0.000..0.000 rows=4 loops=1000)
         ->  Limit (actual time=0.005..0.006 rows=4 loops=1)
               ->  Seq Scan on some_table (actual time=0.003..0.004 rows=4 loops=1)
 Planning time: 2.169 ms
 Execution time: 2.059 ms
(110 rows)
```

- `NestLoop` **(вложенный цикл) с секционированной таблицей**, которая опущена здесь, так как была показана выше.

Узнать больше о нестандартных узлах плана вы можете в *блоге* Александра Короткова.

F.36.5. Справка

F.36.5.1. Переменные GUC

Для включения/отключения модуля `pg_pathman` и отдельных узлов плана предназначены несколько переменных GUC:

- `pg_pathman.enable` — включает (отключает) модуль `pg_pathman`.

  По умолчанию: `on` (вкл.)
- `pg_pathman.enable_runtimeappend` — включает нестандартный узел `RuntimeAppend`.

  По умолчанию: `on` (вкл.)
- `pg_pathman.enable_runtimemergeappend` — включает нестандартный узел `RuntimeMergeAppend`.

  По умолчанию: `on` (вкл.)
- `pg_pathman.enable_partitionfilter` — включает нестандартный узел `PartitionFilter`, выполняющий межсекционные операции `INSERT`.

  По умолчанию: `on` (вкл.)
- `pg_pathman.enable_partitionrouter` — включает нестандартный узел `PartitionRouter`, выполняющий межсекционные операции `UPDATE`.

  По умолчанию: `off` (выкл.)

- pg_pathman.enable_auto_partition — включает автоматическое создание секций (в рамках сеанса).

  По умолчанию: on (вкл.)
- pg_pathman.enable_bounds_cache — включает/отключает кеш границ.

  По умолчанию: on (вкл.)
- pg_pathman.insert_into_fdw — разрешает использовать операции INSERT с различными обёртками сторонних данных. Возможные значения: disabled (такое использование запрещено), postgres (разрешено для обёртки postgres) и any_fdw (разрешено для любых обёрток).

  По умолчанию: postgres
- pg_pathman.override_copy — включает/отключает перехват оператора COPY.

  По умолчанию: on (вкл.)

F.36.5.2. Представления и таблицы

F.36.5.2.1. pathman_config

В этой таблице хранится список секционированных таблиц. Это основное хранилище конфигурации.

```
CREATE TABLE IF NOT EXISTS pathman_config (
    partrel         REGCLASS NOT NULL PRIMARY KEY,
    attname         TEXT NOT NULL,
    parttype        INTEGER NOT NULL,
    range_interval  TEXT);
```

F.36.5.2.2. pathman_config_params

В этой таблице хранятся дополнительные параметры, переопределяющие стандартное поведение pg_pathman.

```
CREATE TABLE IF NOT EXISTS pathman_config_params (
    partrel        REGCLASS NOT NULL PRIMARY KEY,
    enable_parent  BOOLEAN NOT NULL DEFAULT TRUE,
    auto           BOOLEAN NOT NULL DEFAULT TRUE,
    init_callback  REGPROCEDURE NOT NULL DEFAULT 0,
    spawn_using_bgw BOOLEAN NOT NULL DEFAULT FALSE);
```

F.36.5.2.3. pathman_concurrent_part_tasks

В этом представлении показываются все работающие в данный момент задачи параллельного разбиения.

```
-- вспомогательная функция, возвращающая множество
CREATE OR REPLACE FUNCTION show_concurrent_part_tasks()
RETURNS TABLE (
    userid     REGROLE,
    pid        INT,
    dbid       OID,
    relid      REGCLASS,
    processed  INT,
    status     TEXT)
AS 'pg_pathman', 'show_concurrent_part_tasks_internal'
LANGUAGE C STRICT;

CREATE OR REPLACE VIEW pathman_concurrent_part_tasks
AS SELECT * FROM show_concurrent_part_tasks();
```

F.36.5.2.4. pathman_partition_list

В этом представлении показываются все существующие разделы, а также их родители и границы диапазонов (NULL для хеш-секций).

```
-- вспомогательная функция, возвращающая множество
CREATE OR REPLACE FUNCTION show_partition_list()
RETURNS TABLE (
    parent     REGCLASS,
    partition  REGCLASS,
    parttype   INT4,
    expr       TEXT,
    range_min  TEXT,
    range_max  TEXT)
AS 'pg_pathman', 'show_partition_list_internal'
LANGUAGE C STRICT;

CREATE OR REPLACE VIEW pathman_partition_list
AS SELECT * FROM show_partition_list();
```

F.36.5.3. Функции

F.36.5.3.1. Функции для создания секций

```
create_hash_partitions(parent_relid     REGCLASS,
                       expression       TEXT,
                       partitions_count INTEGER,
                       partition_data   BOOLEAN DEFAULT TRUE,
                       partition_names  TEXT[] DEFAULT NULL,
                       tablespaces      TEXT[] DEFAULT NULL)
```

Выполняет секционирование по хешу для таблицы `relation` по целочисленному ключу `expression`. Параметр `partitions_count` задаёт число создаваемых секций; он не может быть изменён впоследствии. Если параметр `partition_data` равен `true`, все данные будут автоматически переноситься из родительской таблицы в секции. Заметьте, что перенос данных может занять некоторое время, и таблица будет заблокирована до завершения транзакции. Если вам нужно перенести данные без блокировки, воспользуйтесь функцией `partition_table_concurrently()`. Для каждой секции вызывается обработчик создания секции, если он был установлен заранее (см. `set_init_callback()`).

```
create_range_partitions(relation       REGCLASS,
                        expression     TEXT,
                        start_value    ANYELEMENT,
                        p_interval     ANYELEMENT,
                        p_count        INTEGER DEFAULT NULL,
                        partition_data BOOLEAN DEFAULT TRUE)

create_range_partitions(relation       REGCLASS,
                        expression     TEXT,
                        start_value    ANYELEMENT,
                        p_interval     INTERVAL,
                        p_count        INTEGER DEFAULT NULL,
                        partition_data BOOLEAN DEFAULT TRUE)

create_range_partitions(relation        REGCLASS,
                        expression      TEXT,
                        bounds          ANYARRAY,
                        partition_names TEXT[] DEFAULT NULL,
                        tablespaces     TEXT[] DEFAULT NULL,
                        partition_data  BOOLEAN DEFAULT TRUE)
```

Выполняет секционирование по диапазонам таблицы relation по ключу разбиения expression. Аргумент start_value задаёт начальное значение, p_interval задаёт диапазон по умолчанию для автоматически создаваемых секций или секций, создаваемых функциями append_range_partition() или prepend_range_partition(). Если в p_interval передаётся NULL, автоматическое создание секций отключается. В p_count задаётся число заранее создаваемых секций (если p_count не задано, pg_pathman пытается определить число секций по значению expression). Массив bounds определяет границы секций, которые должны быть созданы. Этот массив можно построить, воспользовавшись функцией generate_range_bounds(). Для каждой секции вызывается обработчик создания секции, если он был установлен заранее.

F.36.5.3.2. Функции для переноса данных

```
partition_table_concurrently(relation REGCLASS,
                             batch_size INTEGER DEFAULT 1000,
                             sleep_time FLOAT8 DEFAULT 1.0)
```

Запускает фоновый рабочий процесс для переноса данных из родительской таблицы в секции. Этот рабочий процесс копирует данные в коротких транзакциях небольшими блоками (до 10000 строк в транзакции) и поэтому не оказывает значительного влияния на работу пользователей.

```
stop_concurrent_part_task(relation REGCLASS)
```

Останавливает фоновый рабочий процесс, выполняющий задачу параллельного секционирования. Замечание: рабочий процесс завершается после того, как заканчивает перемещение текущей порции данных.

F.36.5.3.3. Триггеры

Для операций INSERT и межсекционных UPDATE триггеры более не требуются. При этом пользовательские триггеры поддерживаются:

- Каждое добавление строки приводит к выполнению триггерных функций BEFORE/AFTER INSERT для соответствующей секции.
- Каждое изменение строки приводит к выполнению триггерных функций BEFORE/AFTER UPDATE для соответствующей секции.
- Каждое перемещение строки (межсекционное изменение) приводит к выполнению триггерных функций BEFORE UPDATE + BEFORE/AFTER DELETE + BEFORE/AFTER INSERT для соответствующих секций.

F.36.5.3.4. Функции для управления секциями

```
replace_hash_partition(old_partition       REGCLASS,
                       new_partition       REGCLASS,
                       lock_parent         BOOLEAN DEFAULT TRUE)
```

Заменяет заданную секцию таблицы, секционированной по хешу, другой таблицей. Если lock_parent имеет значение true, операции INSERT/UPDATE/ALTER TABLE в родительской таблице не допускаются.

```
split_range_partition(partition_relid  REGCLASS,
                      split_value      ANYELEMENT,
                      partition_name   TEXT DEFAULT NULL,
                      tablespace       TEXT DEFAULT NULL)
```

Разбивает диапазонную секцию partition на две по значению value. Для создаваемой секции вызывается обработчик создания секции, если он задан.

```
merge_range_partitions(variadic partitions REGCLASS[])
```

Объединяет несколько соседних секций при разбиении по диапазонам. Секции автоматически упорядочиваются по расположению диапазонов. Все данные будут собраны в первой секции, а остальные секции будут удалены. Если остающаяся секция содержит дочерние секции, для добавляемых при объединении данных могут создаваться новые дочерние секции согласно существующему выражению разбиения.

```
append_range_partition(parent_relid   REGCLASS,
                       partition_name TEXT DEFAULT NULL,
                       tablespace     TEXT DEFAULT NULL)
```

Добавляет новую диапазонную секцию с интервалом `pathman_config.range_interval` в конец списка секций.

```
prepend_range_partition(parent_relid   REGCLASS,
                        partition_name TEXT DEFAULT NULL,
                        tablespace     TEXT DEFAULT NULL)
```

Добавляет новую диапазонную секцию с интервалом `pathman_config.range_interval` в начало списка секций.

```
add_range_partition(parent_relid   REGCLASS,
                    start_value    ANYELEMENT,
                    end_value      ANYELEMENT,
                    partition_name TEXT DEFAULT NULL,
                    tablespace     TEXT DEFAULT NULL)
```

Добавляет новую диапазонную секцию для таблицы `relation` с заданными границами диапазона. Если в качестве `start_value` или `end_value` передаётся NULL, соответствующая граница диапазона будет бесконечной.

```
drop_range_partition(partition_relid TEXT, delete_data BOOLEAN DEFAULT TRUE)
```

Удаляет диапазонную секцию, а также все содержащиеся в ней данные, если установлен флаг `delete_data`.

```
attach_range_partition(parent_relid    REGCLASS,
                       partition_relid REGCLASS,
                       start_value     ANYELEMENT,
                       end_value       ANYELEMENT)
```

Присоединяет секцию к существующему отношению с секционированием по диапазонам. Структура присоединяемой таблицы должна в точности повторять структуру родительской, включая удалённые столбцы. Если установлен, вызывается обработчик создания секции (см. Подраздел F.36.5.2.2).

```
detach_range_partition(partition_relid REGCLASS)
```

Отсоединяет секцию от существующего отношения с секционированием по диапазонам.

```
disable_pathman_for(parent_relid REGCLASS)
```

Полностью отключает механизм секционирования `pg_pathman` для заданной родительской таблицы и удаляет триггер на добавление, если он существует. При этом все секции и данные остаются без изменений.

```
drop_partitions(parent_relid REGCLASS,
                delete_data  BOOLEAN DEFAULT FALSE)
```

Удаляет секции родительской таблицы (как сторонние, так и локальные). Если параметр `delete_data` равен `false` (это значение по умолчанию), данные сначала копируются в родительскую таблицу.

F.36.5.3.5. Дополнительные функции

```
pathman_version()
```

Возвращает номер версии `pg_pathman`.

```
set_interval(relation REGCLASS, value ANYELEMENT)
```

Изменяет интервал для таблицы, секционированной по диапазонам. Заметьте, что этот интервал должен быть неотрицательными и он не должен быть пустым, то есть его значение должно быть

больше нуля для числовых типов, не меньше 1 микросекунды для типа `timestamp` и не меньше 1 дня для типа `date`.

`set_enable_parent(relation REGCLASS, value BOOLEAN)`

Включает/исключает родительскую таблицу в план запроса. В оригинальном планировщике Postgres Pro родительская таблица всегда включается в план запроса, даже если она пуста, что может повлечь дополнительные издержки. Вы можете исключить родительскую таблицу из рассмотрения, если не собираетесь никогда хранить в ней какие-либо данные. Значение по умолчанию зависит от параметра `partition_data`, заданного при изначальном создании секций функцией `create_range_partitions()`. Если параметр `partition_data` имел значение `true`, значит все данные уже были перенесены в секции, и родительская таблица отключается. В противном случае она включена.

`set_auto(relation REGCLASS, value BOOLEAN)`

Включает/отключает автоматическое создание секций (только для секционирования по диапазонам). По умолчанию этот режим включён.

`set_init_callback(relation REGCLASS, callback REGPROCEDURE DEFAULT 0)`

Устанавливает обработчик создания секции, который будет вызываться для каждой присоединяемой или создаваемой секции (по диапазонам или по хешу). Если обработчик имеет характеристику `SECURITY INVOKER`, он выполняется от имени пользователя, который выполнял оператор, требующий создания новой секции. Например:

`INSERT INTO partitioned_table VALUES (-5)`

Обработчик должен иметь следующую сигнатуру: `part_init_callback(args JSONB) RETURNS VOID`. В параметре `arg` передаются несколько полей, зависящих от типа секционирования:

```
/* Таблица abc с секционированием по диапазонам (потомок abc_4) */
{
    "parent":    "abc",
    "parttype":  "2",
    "partition": "abc_4",
    "range_max": "401",
    "range_min": "301"
}

/* Таблица abc с секционированием по хешу (потомок abc_0) */
{
    "parent":    "abc",
    "parttype":  "1",
    "partition": "abc_0"
}
```

`set_spawn_using_bgw(relation REGCLASS, value BOOLEAN)`

Включает использование SpawnPartitionsWorker для создания новых секций в отдельной транзакции в случае добавления данных за пределами диапазона разбиения.

`create_naming_sequence(parent_relid REGCLASS)`

Включает автоматическое именование секций для заданной таблицы `relation`. Эту функцию необходимо использовать при разбиении таблицы по составному ключу.

```
add_to_pathman_config(parent_relid     REGCLASS,
                      expression       TEXT,
                      range_interval   TEXT)
add_to_pathman_config(parent_relid     REGCLASS,
```

```
                      expression       TEXT)
```

Регистрирует указанную таблицу `relation` в `pg_pathman` для выполнения секционирования по заданному выражению (`expression`). Для разбиения по диапазонам аргумент `range_interval` является обязательным. Вы можете передать в нём `NULL`, если планируете добавлять секции вручную.

```
generate_range_bounds(p_start     ANYELEMENT,
                      p_interval  INTERVAL,
                      p_count     INTEGER)

generate_range_bounds(p_start     ANYELEMENT,
                      p_interval  ANYELEMENT,
                      p_count     INTEGER)
```

Строит массив `bounds` с границами секций, которые должны быть созданы. Этот массив можно передать в качестве аргумента функции `create_range_partitions()`.

F.36.6. Авторы

- Ильдар Мусин
- Александр Коротков
- Дмитрий Иванов

F.37. pg_prewarm

Модуль `pg_prewarm` предоставляет удобную возможность загружать данные отношений в кеш операционной системы или в кеш буферов Postgres Pro.

F.37.1. Функции

```
pg_prewarm(regclass, mode text default 'buffer', fork text default 'main',
           first_block int8 default null,
           last_block int8 default null) RETURNS int8
```

Первый аргумент задаёт отношение, которое будет «разогрето». Во втором указывается метод «разогрева», из описанных ниже; в третьем задаётся целевой слой отношения, обычно `main`. В четвёртом аргументе можно передать номер первого разогреваемого блока (`NULL` принимается как синоним нуля), а в пятом — номер последнего блока (`NULL` означает последний блок отношения). Возвращает эта функция количество разогретых блоков.

Эта функция поддерживает три режима разогрева. В режиме `prefetch` выдаются асинхронные запросы предвыборки данных операционной системе, если они поддерживаются, либо происходит ошибка. В режиме `read` считывается заданный диапазон блоков; в отличие от `prefetch` это происходит синхронно и поддерживается во всех ОС и любыми сборками, но может быть медленнее. В режиме `buffer` запрошенный диапазон блоков считывается в кеш буферов базы данных.

Заметьте, что с любым из этих методов попытка разогреть больше блоков, чем может уместиться в кеше (в кеше ОС в режимах `prefetch` и `read`, либо в кеше Postgres Pro в режиме `buffer`) скорее всего приведёт к тому, что блоки с меньшими номерами будут вытеснены из кеша при чтении последующих блоков. Кроме того, разогретые данные никаким специальным образом не защищаются от вытеснения из кеша, так что возможна ситуация, когда из-за другой активности только что разогретые блоки будут вытеснены вскоре после чтения; с другой стороны при таком разогреве из кеша могут быть вытеснены другие данные. Поэтому разогрев обычно наиболее полезен при загрузке, когда кеши в основном пусты.

F.37.2. Автор

Роберт Хаас <rhaas@postgresql.org>

F.38. pg_query_state

Модуль pg_query_state даёт возможность узнавать текущее состояние выполнения запросов в работающем обслуживающем процессе.

F.38.1. Обзор

Каждый обычный (не служебный) оператор запроса (SELECT/INSERT/UPDATE/DELETE) после этапа оптимизации/планирования переводится в дерево плана, представляющего собой своего рода императивное представление SQL-запроса. Команда EXPLAIN ANALYZE позволяет просмотреть статистику выполнения, собранную с каждого узла этого дерева плана (общее время выполнения, число строк, выданных верхних узлам и т. д.). Но эта статистика собирается только после выполнения запроса. Данный модуль позволяет просмотреть актуальную статистику запроса, работающего во внешнем обслуживающем процессе. При этом формат результирующего вывода практически идентичен выводу обычной команды EXPLAIN ANALYZE. Таким образом, пользователи могут отслеживать собственно процесс выполнения запросов. В действительности, этот модуль может исследовать внешний обслуживающий процесс и определять его фактическое состояние. В частности, это полезно, когда обслуживающий процесс выполняет очень сложный запрос или «завис».

F.38.2. Сценарии использования

С помощью этого модуля возможно:

- выявлять сложные запросы (в совокупности с другими средствами мониторинга);
- наблюдать за выполнением запросов.

F.38.3. Установка

Чтобы установить pg_query_state, выполните в psql:

```
CREATE EXTENSION pg_query_state;
```

Затем модифицируйте параметр shared_preload_libraries в конфигурационном файле postgres.conf:

```
shared_preload_libraries = 'pg_query_state'
```

Для вступления изменений в силу потребуется перезапустить экземпляр Postgres Pro.

F.38.4. Функция pg_query_state

```
pg_query_state(integer  pid,
               verbose  boolean DEFAULT FALSE,
               costs    boolean DEFAULT FALSE,
               timing   boolean DEFAULT FALSE,
               buffers  boolean DEFAULT FALSE,
               triggers boolean DEFAULT FALSE,
               format   text    DEFAULT 'text')
  returns TABLE (pid integer,
                 frame_number integer,
                 query_text text,
                 plan text,
                 leader_pid integer)
```

Получает от обслуживающего процесса с заданным pid состояние текущего запроса. При выполнении параллельного запроса может порождаться множество параллельных запросов, а при вызовах функций образуются вложенные подзапросы, так что состояние выполнения может представляться в виде стека запросов, поэтому данная функция возвращает результат типа TABLE (pid integer, frame_number integer, query_text text, plan text, leader_pid integer).

Он представляет структуру дерева, состоящего ведущий процесс и порождённые им рабочие процессы, идентифицируемые по `pid`. Каждый рабочий процесс ссылается на свой ведущий по столбцу `leader_pid`. Для ведущего процесса значение этого столбца — `NULL`. Состояние каждого процесса представляется в виде стека вызовов функций. Для каждого кадра этого стека устанавливается соответствие по столбцам `frame_number` (номер кадра, начиная с нуля), `query_text` (текст запроса) и `plan` (план) со столбцами текущей статистики.

Таким образом, пользователь может видеть состояние основного запроса и запросов, порождённых вызовами функции, для ведущего процесса и всех рабочих процессов, порождённых им.

В процессе выполнения некоторые узлы плана могут представлять циклы. Поэтому статистика по такому узлу состоит из двух частей: средняя статистика по предыдущим циклам, как в выводе EXPLAIN ANALYZE, и статистика для текущего, ещё не законченного цикла.

Необязательные аргументы:

- verbose — использовать EXPLAIN VERBOSE для вывода плана;
- costs — стоимости каждого узла;
- timing — выводить время исполнения для каждого узла; если сбор временной статистики на вызываемой стороне отключён, в результате будет выдаваться ПРЕДУПРЕЖДЕНИЕ `timing statistics disabled`;
- buffers — выводить использование буферов; если сбор статистики буферов на вызываемой стороне отключён, в результате будет выдаваться ПРЕДУПРЕЖДЕНИЕ `buffers statistics disabled`;
- triggers — включать в деревья планов статистику выполнения триггеров;
- format — задаёт формат EXPLAIN для вывода планов; возможные значения: text, xml, json и yaml.

Если вызываемый обслуживающий процесс не выполняет запросы, данная функция выводит сообщение о его состоянии, полученном из представления `pg_stat_activity`, если там есть информация о нём.

Вызывать эту функцию разрешено только суперпользователям или членам роли, которой принадлежит обслуживающий процесс. В противном случае функция печатает сообщение `permission denied`.

F.38.5. Конфигурационные параметры

Для управления всем модулем и сбора определённых статистических показателей в процессе выполнения запроса доступен также набор доступных пользователю переменных GUC:

- pg_query_state.enable — отключить (или включить) pg_query_state полностью, значение по умолчанию: `true`
- pg_query_state.enable_timing — собирать статистику времени по каждому узлу, значение по умолчанию: `false`
- pg_query_state.enable_buffers — собирать статистику по использованию буферов; значение по умолчанию: `false`

Эти параметры должны устанавливаться на вызываемой стороне до выполнения запросов, статистика которых представляет интерес. Если параметр pg_query_state.enable_timing отключён, вызывающая сторона не может получать статистику по времени и выдаётся `ПРЕДУПРЕЖДЕНИЕ`; аналогично с параметром pg_query_state.enable_buffers.

F.38.6. Примеры

Установить максимальное число параллельных исполнителей для узла `Gather` равным 2:

```
postgres=# set max_parallel_workers_per_gather = 2;
```

Предположим, что один обслуживающий процесс с `pid` = 49265 выполняет простой запрос:

```
postgres=# select pg_backend_pid();
```

```
pg_backend_pid
----------------
    49265
(1 row)
postgres=# select count(*) from foo join bar on foo.c1=bar.c1;
```

Другие обслуживающие процессы могут извлекать промежуточное состояние о выполнении этого запроса:

```
postgres=# \x
postgres=# select * from pg_query_state(49265);
-[ RECORD
 1 ]+-----------------------------------------------------------------------------------------
pid          | 49265
frame_number | 0
query_text   | select count(*) from foo join bar on foo.c1=bar.c1;
plan         | Finalize Aggregate (Current loop: actual rows=0, loop number=1)
                                          +
             |   ->  Gather (Current loop: actual rows=0, loop number=1)
                                          +
             |         Workers Planned: 2
                                          +
             |         Workers Launched: 2
                                          +
             |         ->  Partial Aggregate (Current loop: actual rows=0, loop
 number=1)                                  +
             |               ->  Nested Loop (Current loop: actual rows=12, loop
 number=1)                                 +
             |                     Join Filter: (foo.c1 = bar.c1)
                                          +
             |                     Rows Removed by Join Filter: 5673232
                                          +
             |                     ->  Parallel Seq Scan on foo (Current loop: actual
 rows=12, loop number=1)                    +
             |                     ->  Seq Scan on bar (actual rows=500000 loops=11)
 (Current loop: actual rows=173244, loop number=12)
leader_pid   | (null)
-[ RECORD
 2 ]+-----------------------------------------------------------------------------------------
pid          | 49324
frame_number | 0
query_text   | <parallel query>
plan         | Partial Aggregate (Current loop: actual rows=0, loop number=1)
                                          +
             |   ->  Nested Loop (Current loop: actual rows=10, loop number=1)
                                          +
             |         Join Filter: (foo.c1 = bar.c1)
                                          +
             |         Rows Removed by Join Filter: 4896779
                                          +
             |         ->  Parallel Seq Scan on foo (Current loop: actual rows=10, loop
 number=1)                                +
             |         ->  Seq Scan on bar (actual rows=500000 loops=9) (Current loop:
 actual rows=396789, loop number=10)
leader_pid   | 49265
-[ RECORD
 3 ]+-----------------------------------------------------------------------------------------
pid          | 49323
```

```
frame_number | 0
query_text   | <parallel query>
plan         | Partial Aggregate (Current loop: actual rows=0, loop number=1)
                                        +
             |   ->  Nested Loop (Current loop: actual rows=11, loop number=1)
                                        +
             |         Join Filter: (foo.c1 = bar.c1)
                                        +
             |         Rows Removed by Join Filter: 5268783
                                        +
             |         ->  Parallel Seq Scan on foo (Current loop: actual rows=11, loop
 number=1)                              +
             |         ->  Seq Scan on bar (actual rows=500000 loops=10) (Current loop:
 actual rows=268794, loop number=11)
leader_pid   | 49265
```

В показанном выше примере обслуживающий процесс порождает два параллельных рабочих процесса с PID 49324 и 49323. Их значения столбца `leader_pid` показывают, что эти рабочие процессы относятся к основном ведущему. Узел `Seq Scan` показывает статистику по выполненным циклам (показывается среднее число строк, переданное узлу `Nested Loop`, и число выполненных циклов) и статистику по текущему узлу. Другие узлы показывают статистику только по текущему циклу, так как это первая итерация (номер цикла = 1).

Предположим, что первый обслуживающий процесс выполняет некоторую функцию:

```
postgres=# select n_join_foo_bar();
```

Другой процесс может получить такую информацию:

```
postgres=# select * from pg_query_state(49265);
-[ RECORD
 1 ]+-----------------------------------------------------------------------------------------
pid          | 49265
frame_number | 0
query_text   | select n_join_foo_bar();
plan         | Result (Current loop: actual rows=0, loop number=1)
leader_pid   | (null)
-[ RECORD
 2 ]+-----------------------------------------------------------------------------------------
pid          | 49265
frame_number | 1
query_text   | SELECT (select count(*) from foo join bar on foo.c1=bar.c1)
plan         | Result (Current loop: actual rows=0, loop number=1)
                                      +
             |   InitPlan 1 (returns $0)
                                      +
             |     ->  Aggregate (Current loop: actual rows=0, loop number=1)
                                      +
             |           ->  Nested Loop (Current loop: actual rows=51, loop number=1)
                                      +
             |                 Join Filter: (foo.c1 = bar.c1)
                                      +
             |                 Rows Removed by Join Filter: 51636304
                                      +
             |                 ->  Seq Scan on bar (Current loop: actual rows=52, loop
 number=1)                             +
             |                 ->  Materialize (actual rows=1000000 loops=51) (Current
 loop: actual rows=636355, loop number=52)+
```

```
             |                        ->  Seq Scan on foo (Current loop: actual
 rows=1000000, loop number=1)
leader_pid   | (null)
```

Первая строка соответствует вызову функции, вторая — запросу, содержащемуся в теле этой функции.

Мы можем получать результирующие планы в различных форматах (например, в json):

```
postgres=# select * from pg_query_state(pid := 49265, format := 'json');
-[ RECORD 1 ]+------------------------------------------------------------
pid          | 49265
frame_number | 0
query_text   | select * from n_join_foo_bar();
plan         | {                                                          +
             |   "Plan": {                                                +
             |     "Node Type": "Function Scan",                          +
             |     "Parallel Aware": false,                               +
             |     "Function Name": "n_join_foo_bar",                     +
             |     "Alias": "n_join_foo_bar",                             +
             |     "Current loop": {                                      +
             |       "Actual Loop Number": 1,                             +
             |       "Actual Rows": 0                                     +
             |     }                                                      +
             |   }                                                        +
             | }
leader_pid   | (null)
-[ RECORD 2 ]+------------------------------------------------------------
pid          | 49265
frame_number | 1
query_text   | SELECT (select count(*) from foo join bar on foo.c1=bar.c1)
plan         | {                                                          +
             |   "Plan": {                                                +
             |     "Node Type": "Result",                                 +
             |     "Parallel Aware": false,                               +
             |     "Current loop": {                                      +
             |       "Actual Loop Number": 1,                             +
             |       "Actual Rows": 0                                     +
             |     },                                                     +
             |     "Plans": [                                             +
             |       {                                                    +
             |         "Node Type": "Aggregate",                          +
             |         "Strategy": "Plain",                               +
             |         "Partial Mode": "Simple",                          +
             |         "Parent Relationship": "InitPlan",                 +
             |         "Subplan Name": "InitPlan 1 (returns $0)",         +
             |         "Parallel Aware": false,                           +
             |         "Current loop": {                                  +
             |           "Actual Loop Number": 1,                         +
             |           "Actual Rows": 0                                 +
             |         },                                                 +
             |         "Plans": [                                         +
             |           {                                                +
             |             "Node Type": "Nested Loop",                    +
             |             "Parent Relationship": "Outer",                +
             |             "Parallel Aware": false,                       +
             |             "Join Type": "Inner",                          +
             |             "Current loop": {                              +
```

```
               |                "Actual Loop Number": 1,                    +
               |                "Actual Rows": 610                          +
               |              },                                            +
               |              "Join Filter": "(foo.c1 = bar.c1)",           +
               |              "Rows Removed by Join Filter": 610072944,     +
               |              "Plans": [                                    +
               |                {                                           +
               |                  "Node Type": "Seq Scan",                  +
               |                  "Parent Relationship": "Outer",           +
               |                  "Parallel Aware": false,                  +
               |                  "Relation Name": "bar",                   +
               |                  "Alias": "bar",                           +
               |                  "Current loop": {                         +
               |                    "Actual Loop Number": 1,                +
               |                    "Actual Rows": 611                      +
               |                  }                                         +
               |                },                                          +
               |                {                                           +
               |                  "Node Type": "Materialize",               +
               |                  "Parent Relationship": "Inner",           +
               |                  "Parallel Aware": false,                  +
               |                  "Actual Rows": 1000000,                   +
               |                  "Actual Loops": 610,                      +
               |                  "Current loop": {                         +
               |                    "Actual Loop Number": 611,              +
               |                    "Actual Rows": 73554                    +
               |                  },                                        +
               |                  "Plans": [                                +
               |                    {                                       +
               |                      "Node Type": "Seq Scan",              +
               |                      "Parent Relationship": "Outer",       +
               |                      "Parallel Aware": false,              +
               |                      "Relation Name": "foo",               +
               |                      "Alias": "foo",                       +
               |                      "Current loop": {                     +
               |                        "Actual Loop Number": 1,            +
               |                        "Actual Rows": 1000000              +
               |                      }                                     +
               |                    }                                       +
               |                  ]                                         +
               |                }                                           +
               |              ]                                             +
               |            }                                               +
               |          ]                                                 +
               |        }                                                   +
               |      ]                                                     +
               |    }                                                       +
               |  }
leader_pid     |  (null)
```

F.39. pgrowlocks

Модуль `pgrowlocks` предоставляет функцию, показывающую информацию о блокировке строк для заданной таблицы.

По умолчанию его использование разрешено суперпользователям, членам роли `pg_stat_scan_tables` и пользователям с правом `SELECT` в заданной таблице.

F.39.1. Обзор

```
pgrowlocks(text) returns setof record
```

В параметре передаётся имя таблицы. В результате возвращается набор записей, в котором строка соответствует строке, заблокированной в таблице. Столбцы результата показаны в Таблице F.22.

Таблица F.22. Столбцы результата `pgrowlocks`

Имя	Тип	Описание
`locked_row`	`tid`	Идентификатор кортежа (TID) блокированной строки
`locker`	`xid`	Идентификатор блокирующей транзакции или идентификатор мультитранзакции, если это мультитранзакция
`multi`	`boolean`	True, если блокирующий субъект — мультитранзакция
`xids`	`xid[]`	Идентификаторы блокирующих транзакций (больше одной для мультитранзакции)
`modes`	`text[]`	Режим блокирования (больше одного для мультитранзакции), массив со значениями `Key Share`, `Share`, `For No Key Update`, `No Key Update`, `For Update`, `Update`.
`pids`	`integer[]`	Идентификаторы блокирующих обслуживающих процессов (больше одного для мультитранзакции)

Функция `pgrowlocks` запрашивает блокировку `AccessShareLock` для целевой таблицы и считывает строку за строкой для сбора информации о блокировке строк. Это происходит небыстро для большой таблицы. Заметьте, что:

1. Если таблица заблокирована в режиме `ACCESS EXCLUSIVE`, функция `pgrowlocks` будет блокироваться.
2. Функция `pgrowlocks` не гарантирует внутреннюю согласованность результатов. В ходе её выполнения могут быть установлены новые блокировки строк, либо освобождены старые.

Функция `pgrowlocks` не показывает содержимое заблокированных строк. Если вы хотите параллельно взглянуть на содержимое строк, можно проделать примерно следующее:

```
SELECT * FROM accounts AS a, pgrowlocks('accounts') AS p
  WHERE p.locked_row = a.ctid;
```

Однако учтите, что такой запрос будет очень неэффективным.

F.39.2. Пример вывода

```
=# SELECT * FROM pgrowlocks('t1');
 locked_row | locker | multi | xids  |     modes      |  pids
------------+--------+-------+-------+----------------+--------
 (0,1)      |    609 | f     | {609} | {"For Share"}  | {3161}
 (0,2)      |    609 | f     | {609} | {"For Share"}  | {3161}
 (0,3)      |    607 | f     | {607} | {"For Update"} | {3107}
 (0,4)      |    607 | f     | {607} | {"For Update"} | {3107}
(4 rows)
```

F.39.3. Автор

Тацуо Исии

F.40. pg_stat_statements

Модуль pg_stat_statements предоставляет возможность отслеживать статистику выполнения сервером всех операторов SQL.

Этот модуль нужно загружать, добавив pg_stat_statements в shared_preload_libraries в файле postgresql.conf, так как ему требуется дополнительная разделяемая память. Это значит, что для загрузки или выгрузки модуля необходимо перезапустить сервер.

Когда модуль pg_stat_statements загружается, он отслеживает статистику по всем базам данных на сервере. Для получения и обработки этой статистики этот модуль предоставляет представление pg_stat_statements и вспомогательные функции pg_stat_statements_reset и pg_stat_statements. Эти объекты не доступны глобально, но их можно установить в определённой базе данных, выполнив команду CREATE EXTENSION pg_stat_statements.

F.40.1. Представление pg_stat_statements

Статистика, собираемая модулем, выдаётся через представление с именем pg_stat_statements. Это представление содержит отдельные строки для каждой комбинации идентификатора базы данных, идентификатора пользователя и идентификатора запроса (но в количестве, не превышающем максимальное число различных операторов, которые может отслеживать модуль). Столбцы представления показаны в Таблице F.23.

Таблица F.23. Столбцы pg_stat_statements

Имя	Тип	Ссылки	Описание
userid	oid	pg_authid .oid	OID пользователя, выполнявшего оператор
dbid	oid	pg_database .oid	OID базы данных, в которой выполнялся оператор
queryid	bigint		Внутренний хеш-код, вычисленный по дереву разбора оператора
query	text		Текст, представляющий оператор
calls	bigint		Число выполнений
total_time	double precision		Общее время, затраченное на оператор, в миллисекундах
min_time	double precision		Минимальное время, затраченное на оператор, в миллисекундах
max_time	double precision		Максимальное время, затраченное на оператор, в миллисекундах

Имя	Тип	Ссылки	Описание
mean_time	double precision		Среднее время, затраченное на оператор, в миллисекундах
stddev_time	double precision		Стандартное отклонение времени, затраченного на оператор, в миллисекундах
rows	bigint		Общее число строк, полученных или затронутых оператором
shared_blks_hit	bigint		Общее число попаданий разделяемых блоков в кеш для данного оператора
shared_blks_read	bigint		Общее число разделяемых блоков, прочитанных данным оператором
shared_blks_dirtied	bigint		Общее число разделяемых блоков, «загрязнённых» данным оператором
shared_blks_written	bigint		Общее число разделяемых блоков, записанных данным оператором
local_blks_hit	bigint		Общее число попаданий локальных блоков в кеш для данного оператора
local_blks_read	bigint		Общее число локальных блоков, прочитанных данным оператором
local_blks_dirtied	bigint		Общее число локальных блоков, «загрязнённых» данным оператором
local_blks_written	bigint		Общее число локальных блоков, записанных данным оператором
temp_blks_read	bigint		Общее число временных блоков, прочитанных данным оператором
temp_blks_written	bigint		Общее число временных блоков,

Имя	Тип	Ссылки	Описание
			записанных данным оператором
blk_read_time	double precision		Общее время, затраченное оператором на чтение блоков, в миллисекундах (если включён track_io_timing, или ноль в противном случае)
blk_write_time	double precision		Общее время, затраченное оператором на запись блоков, в миллисекундах (если включён track_io_timing, или ноль в противном случае)

По соображениям безопасности только суперпользователям и членам роли pg_read_all_stats разрешено видеть текст SQL и queryid запросов, выполняемых другими пользователями. Однако другие пользователи могут видеть статистику, если это представление установлено в их базу данных.

Планируемые запросы (то есть SELECT, INSERT, UPDATE и DELETE) объединяются в одну запись в pg_stat_statements, когда они имеют идентичные структуры запросов согласно внутреннему вычисленному хешу. Обычно два запроса будут считаться равными при таком сравнении, если они семантически равнозначны, не считая значений констант, фигурирующих в запросе. Однако служебные команды (то есть все другие команды) сравниваются строго по текстовым строкам запросов.

Когда значение константы игнорируется в целях сравнения запроса с другими запросами, эта константа заменяется в выводе pg_stat_statements обозначением параметра, например, $1. В остальном этот вывод содержит текст первого запроса, хеш которого равнялся значению queryid, связанному с записью в pg_stat_statements.

В некоторых случаях запросы с визуально различными текстами могут быть объединены в одну запись pg_stat_statements. Обычно это происходит только для семантически равнозначных запросов, но есть небольшая вероятность, что из-за наложений хеша несвязанные запросы могут оказаться объединёнными в одной записи. (Однако это невозможно для запросов, принадлежащих разным пользователям баз данных.)

Так как значение хеша queryid вычисляется по представлениям запроса на стадии после разбора, возможна и обратная ситуация: запросы с одинаковым текстом могут оказаться в разных записях, если они получили различные представления по разным причинам, например, из-за изменения search_path.

Потребители статистики pg_stat_statements могут пожелать использовать в качестве более стабильного и надёжного идентификатора для каждой записи не текст запроса, а queryid (возможно, в сочетании с dbid и userid). Однако важно понимать, что стабильность значения хеша queryid гарантируется с ограничениями. Так как этот идентификатор получается из дерева запроса после анализа, его значение будет, помимо прочего, зависеть от внутренних идентификаторов объектов, фигурирующих в этом представлении. С этим связано несколько неинтуитивных следствий. Например, pg_stat_statements будет считать два одинаково выглядящих запроса разными, если они обращаются к таблице, которая была удалена, а затем воссоздана между этими запросами. Результат хеширования также чувствителен к различиям в

машинной архитектуре и другим особенностям платформы. Более того, не стоит рассчитывать на то, что queryid будет оставаться неизменным при обновлении основных версий Postgres Pro.

Как правило, значения queryid можно считать надёжными и сравнимыми, только с условием, что версия сервера и детали метаданных каталога неизменны. Следовательно, можно ожидать, что два сервера, участвующие в репликации на основе воспроизведения физического WAL, будут иметь одинаковые queryid для одного запроса. Однако схемы с логической репликацией не гарантируют сохранения идентичности реплик во всех имеющих значение деталях, так что queryid не будет полезным идентификатором для накопления показателей стоимости по набору логических реплик. В случае сомнений в том или ином подходе, рекомендуется непосредственно протестировать его.

Обозначения параметров, применяемые для замены констант в представляющем запросы тексте, нумеруются, начиная со следующего за последним параметром $*n* в исходном тексте запроса, или с $1 в отсутствие параметров в нём. Стоит отметить, что в некоторых случаях на эту нумерацию могут влиять скрытые символы параметров. Например, PL/pgSQL применяет такие символы для добавления в запросы значений локальных переменных функций, так что оператор PL/pgSQL вида SELECT i + 1 INTO j будет представлен в тексте как SELECT i + $2.

Текст, представляющий запрос, сохраняется во внешнем файле на диске и не занимает разделяемую память. Поэтому даже очень объёмный текст запроса может быть сохранён успешно. Однако если в файле накапливается много длинных текстов запросов, он может вырасти до неудобоваримого размера. В качестве решения этой проблемы, pg_stat_statements может решить стереть текст запросов, и в результате во всех существующих записях в представлении pg_stat_statements в поле query окажутся значения NULL, хотя статистика, связанная с каждым queryid будет сохранена. Если это происходит и мешает анализу, возможно, стоит уменьшить pg_stat_statements.max для предотвращения таких ситуаций.

F.40.2. Функции

pg_stat_statements_reset() returns void

Функция pg_stat_statements_reset очищает всю статистику, собранную к этому времени модулем pg_stat_statements. По умолчанию эту функцию могут выполнять только суперпользователи.

pg_stat_statements(showtext boolean) returns setof record

Представление pg_stat_statements реализовано на базе функции, которая тоже называется pg_stat_statements. Клиенты могут вызывать функцию pg_stat_statements непосредственно, и могут указать showtext := false и получить результат без текста запроса (то есть, выходной аргумент (OUT), соответствующий столбцу представления query, будет содержать NULL). Эта возможность предназначена для поддержки внешних инструментов, для которых желательно избежать издержек, связанных с получением текстов запросов неопределённой длины. Такие инструменты могут кешировать текст первого запроса, который они получат самостоятельно, как это и делает pg_stat_statements, а затем запрашивать тексты запросов только при необходимости. Так как сервер сохраняет тексты запросов в файле, этот подход сокращает объём физического ввода/вывода, порождаемого при постоянном обращении к данным pg_stat_statements.

F.40.3. Параметры конфигурации

pg_stat_statements.max (integer)

Параметр pg_stat_statements.max задаёт максимальное число операторов, отслеживаемых модулем (то есть, максимальное число строк в представлении pg_stat_statements). Когда на обработку поступает больше, чем заданное число различных операторов, информация о редко выполняемых операторах отбрасывается. Значение по умолчанию — 5000. Этот параметр можно задать только при запуске сервера.

pg_stat_statements.track (enum)

Параметр pg_stat_statements.track определяет, какие операторы будут отслеживаться модулем. Со значением top отслеживаются операторы верхнего уровня (те, что непосредственно выполняются клиентами), со значением all также отслеживаются вложенные операторы (например, операторы, вызываемые внутри функций), а значение none полностью отключает сбор статистики по операторам. Значение по умолчанию — top. Изменять этот параметр могут только суперпользователи.

pg_stat_statements.track_utility (boolean)

Параметр pg_stat_statements.track_utility определяет, будет ли этот модуль отслеживать служебные команды. Служебными командами считаются команды, отличные от SELECT, INSERT, UPDATE и DELETE. Значение по умолчанию — on (вкл.). Изменить этот параметр могут только суперпользователи.

pg_stat_statements.save (boolean)

Параметр pg_stat_statements.save определяет, должна ли статистика операторов сохраняться после перезагрузки сервера. Если он отключён (имеет значение off), статистика не сохраняется при остановке сервера и не перезагружается при запуске. Значение по умолчанию — on (вкл.). Этот параметр можно задать только в postgresql.conf или в командной строке сервера.

Этому модулю требуется дополнительная разделяемая память в объёме, пропорциональном pg_stat_statements.max. Заметьте, что эта память будет занята при загрузке модуля, даже если pg_stat_statements.track имеет значение none.

Эти параметры должны задаваться в postgresql.conf. Обычное использование выглядит так:

```
# postgresql.conf
shared_preload_libraries = 'pg_stat_statements'

pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

F.40.4. Пример вывода

```
bench=# SELECT pg_stat_statements_reset();

$ pgbench -i bench
$ pgbench -c10 -t300 bench

bench=# \x
bench=# SELECT query, calls, total_time, rows, 100.0 * shared_blks_hit /
               nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
          FROM pg_stat_statements ORDER BY total_time DESC LIMIT 5;
-[ RECORD 1 ]---------------------------------------------------------------------
query       | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid = $2;
calls       | 3000
total_time  | 9609.00100000002
rows        | 2836
hit_percent | 99.9778970000200936
-[ RECORD 2 ]---------------------------------------------------------------------
query       | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid = $2;
calls       | 3000
total_time  | 8015.156
rows        | 2990
hit_percent | 99.9731126579631345
-[ RECORD 3 ]---------------------------------------------------------------------
query       | copy pgbench_accounts from stdin
calls       | 1
```

```
total_time  | 310.624
rows        | 100000
hit_percent | 0.30395136778115501520
-[ RECORD 4 ]-------------------------------------------------------------------
query       | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2;
calls       | 3000
total_time  | 271.741999999997
rows        | 3000
hit_percent | 93.7968855088209426
-[ RECORD 5 ]-------------------------------------------------------------------
query       | alter table pgbench_accounts add primary key (aid)
calls       | 1
total_time  | 81.42
rows        | 0
hit_percent | 34.4947735191637631
```

F.40.5. Авторы

Такахиро Итагаки <itagaki.takahiro@oss.ntt.co.jp>. Нормализацию запросов добавил Питер Гейган <peter@2ndquadrant.com>.

F.41. pgstattuple

Модуль `pgstattuple` предоставляет различные функции для получения статистики на уровне кортежей.

Так как эти функции возвращают подробную информацию, относящуюся к уровню страницы, только суперпользователь имеет право выполнять их (EXECUTE) после установки. После того как эти функции установлены, с помощью команды `GRANT` можно изменить права доступа к этим функциям, чтобы выполнять их могли не только суперпользователи. По умолчанию доступ к ним имеют члены роли `pg_stat_scan_tables`. За подробностями обратитесь к описанию команды GRANT.

F.41.1. Функции

```
pgstattuple(regclass) returns record
```

Функция `pgstattuple` возвращает физическую длину отношения, процент «мёртвых» кортежей и другую информацию. Она может быть полезна для принятия решения о необходимости очистки. В аргументе передаётся имя (возможно, дополненное схемой) или OID целевого отношения. Например:

```
test=> SELECT * FROM pgstattuple('pg_catalog.pg_proc');
-[ RECORD 1 ]------+-------
table_len          | 458752
tuple_count        | 1470
tuple_len          | 438896
tuple_percent      | 95.67
dead_tuple_count   | 11
dead_tuple_len     | 3157
dead_tuple_percent | 0.69
free_space         | 8932
free_percent       | 1.95
```

Столбцы результата описаны в Таблице F.24.

Таблица F.24. Столбцы результата `pgstattuple`

Столбец	Тип	Описание
`table_len`	`bigint`	Физическая длина отношения в байтах

Столбец	Тип	Описание
tuple_count	bigint	Количество «живых» кортежей
tuple_len	bigint	Общая длина «живых» кортежей в байтах
tuple_percent	float8	Процент «живых» кортежей
dead_tuple_count	bigint	Количество «мёртвых» кортежей
dead_tuple_len	bigint	Общая длина «мёртвых» кортежей в байтах
dead_tuple_percent	float8	Процент «мёртвых» кортежей
free_space	bigint	Общий объём свободного пространства в байтах
free_percent	float8	Процент свободного пространства

Примечание

Значение table_len всегда будет больше суммы tuple_len, dead_tuple_len и free_space. Разница объясняется фиксированными издержками, внутристраничной таблицей указателей на кортежи и пропусками, добавляемыми для выравнивания кортежей.

Функция pgstattuple получает блокировку отношения только для чтения. Таким образом, её результаты отражают не мгновенный снимок; на них будут влиять параллельные изменения.

pgstattuple считает кортеж «мёртвым», если HeapTupleSatisfiesDirty возвращает false.

pgstattuple(text) returns record

Эта функция равнозначна функции pgstattuple(regclass) за исключением того, что для неё целевое отношение задаётся в текстовом виде. Данная функция оставлена для обратной совместимости, в будущем она может перейти в разряд устаревших.

pgstatindex(regclass) returns record

Функция pgstatindex возвращает запись с информацией об индексе типа B-дерево. Например:

```
test=> SELECT * FROM pgstatindex('pg_cast_oid_index');
-[ RECORD 1 ]------+------
version            | 2
tree_level         | 0
index_size         | 16384
root_block_no      | 1
internal_pages     | 0
leaf_pages         | 1
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 54.27
leaf_fragmentation | 0
```

Столбцы результата:

Столбец	Тип	Описание
version	integer	Номер версии B-дерева

Столбец	Тип	Описание
tree_level	integer	Уровень корневой страницы в дереве
index_size	bigint	Общий объём индекса в байтах
root_block_no	bigint	Расположение страницы корня (0, если её нет)
internal_pages	bigint	Количество «внутренних» страниц (верхнего уровня)
leaf_pages	bigint	Количество страниц на уровне листьев
empty_pages	bigint	Количество пустых страниц
deleted_pages	bigint	Количество удалённых страниц
avg_leaf_density	float8	Средняя плотность страниц на уровне листьев
leaf_fragmentation	float8	Фрагментация на уровне листьев

Выдаваемый размер индекса (index_size) обычно вычисляется по формуле internal_pages + leaf_pages + empty_pages + deleted_pages плюс одна страница, так как в нём учитывается и метастраница индекса.

Как и pgstattuple, эта функция собирает данные страница за страницей и не следует ожидать, что её результат представляет мгновенный снимок всего индекса.

pgstatindex(text) returns record

Эта функция равнозначна функции pgstatindex(regclass) за исключением того, что для неё целевое отношение задаётся в текстовом виде. Данная функция оставлена для обратной совместимости, в будущем она может перейти в разряд устаревших.

pgstatginindex(regclass) returns record

Функция pgstatginindex возвращает запись с информацией об индексе типа GIN. Например:

```
test=> SELECT * FROM pgstatginindex('test_gin_index');
-[ RECORD 1 ]--+--
version        | 1
pending_pages  | 0
pending_tuples | 0
```

Столбцы результата:

Столбец	Тип	Описание
version	integer	Номер версии GIN
pending_pages	integer	Количество страниц в списке ожидающих обработки
pending_tuples	bigint	Количество кортежей в списке ожидающих обработки

pgstathashindex(regclass) returns record

Функция pgstathashindex возвращает запись с информацией о хеш-индексе. Например:

```
test=> select * from pgstathashindex('con_hash_index');
-[ RECORD 1 ]--+-----------------
```

```
version        | 4
bucket_pages   | 33081
overflow_pages | 0
bitmap_pages   | 1
unused_pages   | 32455
live_items     | 10204006
dead_items     | 0
free_percent   | 61.8005949100872
```

Столбцы результата:

Столбец	Тип	Описание
version	integer	Номер версии HASH
bucket_pages	bigint	Количество страниц групп
overflow_pages	bigint	Количество страниц переполнения
bitmap_pages	bigint	Количество страниц битовой карты
unused_pages	bigint	Количество неиспользованных страниц
live_items	bigint	Количество «живых» кортежей
dead_tuples	bigint	Количество «мёртвых» кортежей
free_percent	float	Процент свободного пространства

`pg_relpages(regclass) returns bigint`

Функция `pg_relpages` возвращает число страниц в отношении.

`pg_relpages(text) returns bigint`

Эта функция равнозначна функции `pg_relpages(regclass)` за исключением того, что для неё целевое отношение задаётся в текстовом виде. Данная функция оставлена для обратной совместимости, в будущем она может перейти в разряд устаревших.

`pgstattuple_approx(regclass) returns record`

Функция `pgstattuple_approx` является более быстрой альтернативой `pgstattuple`, возвращающей приблизительные результаты. В качестве аргумента ей передаётся имя или OID целевого отношения. Например:

```
test=> SELECT * FROM pgstattuple_approx('pg_catalog.pg_proc'::regclass);
-[ RECORD 1 ]--------+-------
table_len            | 573440
scanned_percent      | 2
approx_tuple_count   | 2740
approx_tuple_len     | 561210
approx_tuple_percent | 97.87
dead_tuple_count     | 0
dead_tuple_len       | 0
dead_tuple_percent   | 0
approx_free_space    | 11996
approx_free_percent  | 2.09
```

Выходные столбцы описаны в Таблице F.25.

Тогда как pgstattuple всегда производит полное сканирование таблицы и возвращает точное число живых и мёртвых кортежей (и их размер), а также точный объём свободного пространства, функция pgstattuple_approx пытается избежать полного сканирования и возвращает точную статистику только по мёртвым кортежам, а количество и объём живых кортежей, как и объём свободного пространства определяет приблизительно.

Она делает это, пропуская страницы, в которых, согласно карте видимости, есть только видимые кортежи (если для страницы установлен соответствующий бит, предполагается, что она не содержит мёртвых кортежей). Для таких страниц эта функция узнаёт объём свободного пространства из карты свободного пространства и предполагает, что остальное пространство на странице занято живыми кортежами.

На страницах, которые нельзя пропустить, она сканирует каждый кортеж, отражает его наличие и размер в соответствующих счётчиках и суммирует свободное пространство на странице. В конце она оценивает приблизительно общее число живых кортежей, исходя из числа просканированных страниц и кортежей (так же, как VACUUM рассчитывает значение pg_class.reltuples).

Таблица F.25. Столбцы результата pgstattuple_approx

Столбец	Тип	Описание
table_len	bigint	Физическая длина отношения в байтах (точная)
scanned_percent	float8	Просканированный процент таблицы
approx_tuple_count	bigint	Количество «живых» кортежей (приблизительное)
approx_tuple_len	bigint	Общая длина «живых» кортежей в байтах (приблизительная)
approx_tuple_percent	float8	Процент «живых» кортежей
dead_tuple_count	bigint	Количество «мёртвых» кортежей (точное)
dead_tuple_len	bigint	Общая длина «мёртвых» кортежей в байтах (точная)
dead_tuple_percent	float8	Процент «мёртвых» кортежей
approx_free_space	bigint	Общий объём свободного пространства в байтах (приблизительный)
approx_free_percent	float8	Процент свободного пространства

В показанном выше выводе показатели свободного пространства могут не соответствовать выводу pgstattuple в точности, потому что карта свободного пространства показывает верное значение, но не гарантируется, что оно будет точным до байта.

F.41.2. Авторы

Тацуо Исии, Сатоши Нагаясу и Абхиджит Менон-Сен

F.42. pg_trgm

Модуль pg_trgm предоставляет функции и операторы для определения схожести алфавитно-цифровых строк на основе триграмм, а также классы операторов индексов, поддерживающие быстрый поиск схожих строк.

F.42.1. Понятия, связанные с триграммами (или триграфами)

Триграмма — это группа трёх последовательных символов, взятых из строки. Мы можем измерить схожесть двух строк, подсчитав число триграмм, которые есть в обеих. Эта простая идея оказывается очень эффективной для измерения схожести слов на многих естественных языках.

> **Примечание**
>
> `pg_trgm`, извлекая триграммы из строк, игнорирует символы, не относящиеся к словам (не алфавитно-цифровые). При выделении триграмм, содержащихся в строке, считается, что перед каждым словом находятся два пробела, а после — один пробел. Например, из строки «`cat`» выделяется набор триграмм: «`  c`», «`  ca`», «`cat`» и «`at `». Из строки «`foo|bar`» выделяются триграммы: «`  f`», «`  fo`», «`foo`», «`oo `», «`  b`», «`  ba`», «`bar`» и «`ar `».

F.42.2. Функции и операторы

Реализованные в модуле `pg_trgm` функции перечислены в Таблице F.26, а операторы — в Таблице F.27.

Таблица F.26. Функции `pg_trgm`

Функция	Возвращает	Описание
`similarity(text, text)`	`real`	Возвращает число, показывающее, насколько близки два аргумента. Диапазон результатов — от нуля (это значение указывает, что две строки полностью различны) до одного (это значение указывает, что две строки идентичны).
`show_trgm(text)`	`text[]`	Возвращает массив всех триграмм в заданной строке. (На практике это редко бывает полезно, кроме как для отладки.)
`word_similarity(text, text)`	`real`	Возвращает число, представляющее наибольшую степень схожести между набором триграмм в первой строке и любым непрерывным фрагментом упорядоченного набора триграмм во второй строке. Подробнее об этом рассказывается ниже.
`show_limit()`	`real`	Возвращает текущий порог схожести, который использует оператор `%`. Это значение задаёт минимальную схожесть между двумя словами, при которой они считаются настолько близкими, что одно может быть, например, ошибочным написанием другого (*устаревшая*).

Функция	Возвращает	Описание
set_limit(real)	real	Задаёт текущий порог схожести, который использует оператор %. Это значение должно быть в диапазоне от 0 до 1 (по умолчанию 0.3). Возвращает то же значение, что было передано на вход (*устаревшая*).

Рассмотрим следующий пример:

```
# SELECT word_similarity('word', 'two words');
 word_similarity
-----------------
             0.8
(1 row)
```

Набор триграмм для первой строки: {" w"," wo","wor","ord","rd "}. Во второй строке упорядоченный набор триграмм: {" t"," tw","two","wo "," w"," wo","wor","ord","rds","ds "}. Наиболее близкий фрагмент упорядоченного множества триграмм во второй строке: {" w"," wo","wor","ord"}, а коэффициент схожести равен 0.8.

Эта функция возвращает значение, которое можно примерно воспринимать как максимальную оценку схожести первой строки с любой подстрокой второй строки. Данная функция не добавляет пробелы к границам фрагмента, поэтому совпадение с отдельным словом оценивается выше, чем совпадение с частью слова.

Таблица F.27. Операторы pg_trgm

Оператор	Возвращает	Описание
text % text	boolean	Возвращает true, если схожесть аргументов выше текущего порога, заданного параметром pg_trgm.similarity_threshold .
text <% text	boolean	Возвращает true, если схожесть между набором триграмм в первом аргументе и непрерывным фрагментом упорядоченного набора триграмм во втором превышает уровень схожести, устанавливаемый параметром pg_trgm.word_similarity_threshold .
text %> text	boolean	Коммутирующий оператор для <%.
text <-> text	real	Возвращает «расстояние» между аргументами, то есть один минус значение similarity() .
text <<-> text	real	Возвращает «расстояние» между аргументами, то есть один минус значение word_similarity() .

Оператор	Возвращает	Описание
text <-> text	real	Коммутирующий оператор для <<->.

F.42.3. Параметры GUC

pg_trgm.similarity_threshold (real)

Задаёт текущий порог схожести, который использует оператор %. Это значение должно быть в диапазоне от 0 до 1 (по умолчанию 0.3).

pg_trgm.word_similarity_threshold (real)

Задаёт текущий порог схожести слов, который используют операторы <% и %>. Это значение должно быть в диапазоне от 0 до 1 (по умолчанию 0.6).

F.42.4. Поддержка индексов

Модуль pg_trgm предоставляет классы операторов индексов GiST и GIN, позволяющие создавать индекс по текстовым столбцам для очень быстрого поиска по критерию схожести. Эти типы индексов поддерживают вышеописанные операторы схожести и дополнительно поддерживают поиск на основе триграмм для запросов с LIKE, ILIKE, ~ и ~*. (Эти индексы не поддерживают простые операторы сравнения и равенства, так что вам может понадобиться и обычный индекс-B-дерево.)

Пример:

```
CREATE TABLE test_trgm (t text);
CREATE INDEX trgm_idx ON test_trgm USING GIST (t gist_trgm_ops);
```

или

```
CREATE INDEX trgm_idx ON test_trgm USING GIN (t gin_trgm_ops);
```

На этот момент у вас будет индекс по столбцу t, используя который можно осуществлять поиск по схожести. Пример типичного запроса:

```
SELECT t, similarity(t, 'слово') AS sml
  FROM test_trgm
  WHERE t % 'слово'
  ORDER BY sml DESC, t;
```

Он выдаст все значения в текстовом столбце, которые достаточно схожи со словом *word*, в порядке сортировки от наиболее к наименее схожим. Благодаря использованию индекса, эта операция будет быстрой даже с очень большими наборами данных.

Другой вариант предыдущего запроса:

```
SELECT t, t <-> 'слово' AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

Он может быть довольно эффективно выполнен с применением индексов GiST, а не GIN. Обычно он выигрышнее первого варианта только когда требуется получить небольшое количество близких совпадений.

Также вы можете использовать индекс по столбцу t для оценки схожести слов. Например:

```
SELECT t, word_similarity('слово', t) AS sml
  FROM test_trgm
  WHERE 'слово' <% t
  ORDER BY sml DESC, t;
```

В результате будут возвращены все значения в текстовом столбце, для которых найдётся непрерывный фрагмент в упорядоченном наборе триграмм, достаточно схожий с набором

триграмм строки *слово*. При этом возвращённые значения будут отсортированы от наиболее к наименее схожим. Этот индекс будет использоваться для ускорения поиска даже с очень большим объёмом данных.

Другой вариант предыдущего запроса:

```
SELECT t, 'слово' <<-> t AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

Он может быть довольно эффективно выполнен с применением индексов GiST, а не GIN.

Начиная с PostgreSQL 9.1, эти типы индексов также поддерживают поиск с операторами `LIKE` и `ILIKE`, например:

```
SELECT * FROM test_trgm WHERE t LIKE '%foo%bar';
```

При таком поиске по индексу сначала из искомой строки извлекаются триграммы, а затем они ищутся в индексе. Чем больше триграмм оказывается в искомой строке, тем более эффективным будет поиск по индексу. В отличие от поиска по B-дереву, искомая строка не должна привязываться к левому краю.

Начиная с PostgreSQL 9.3, индексы этих типов также поддерживают поиск по регулярным выражениям (операторы `~` и `~*`), например:

```
SELECT * FROM test_trgm WHERE t ~ '(foo|bar)';
```

При таком поиске из регулярного выражения извлекаются триграммы, а затем они ищутся в индексе. Чем больше триграмм удаётся извлечь из регулярного выражения, тем более эффективным будет поиск по индексу. В отличие от поиска по B-дереву, искомая строка не должна привязываться к левому краю.

Относительно поиска по регулярному выражению или с `LIKE`, имейте в виду, что при отсутствии триграмм в искомом шаблоне поиск сводится к полному сканирования индекса.

Выбор между индексами GiST и GIN зависит от относительных характеристик производительности GiST и GIN, которые здесь не рассматриваются.

F.42.5. Интеграция с текстовым поиском

Сопоставление триграмм — очень полезный приём в сочетании с применением полнотекстового индекса. В частности это может помочь найти слова, написанные неправильно, которые не будут находиться непосредственно механизмом полнотекстового поиска.

В первую очередь нужно построить дополнительную таблицу, содержащую все уникальные слова в документе:

```
CREATE TABLE words AS SELECT word FROM
        ts_stat('SELECT to_tsvector(''simple'', bodytext) FROM documents');
```

Здесь `documents` — это таблица с текстовым полем `bodytext`, по которому мы будем выполнять поиск. Конфигурация `simple` используется с функцией `to_tsvector` вместо конфигурации для определённого языка по той причине, что нам нужен список исходных (необработанных стеммером) слов.

Затем нужно создать индекс триграмм по столбцу со словами:

```
CREATE INDEX words_idx ON words USING GIN (word gin_trgm_ops);
```

Теперь мы можем использовать запрос `SELECT`, подобный показанному в предыдущем примере, и предлагать варианты исправлений слов, введённых пользователем с ошибками. Кроме того, может быть полезно дополнительно проверить, что выбранные слова также имеют длину, примерно равную длине ошибочных слов.

> **Примечание**
>
> Так как таблица `words` была сформирована как отдельная статическая таблица, её нужно периодически обновлять, чтобы она достаточно хорошо соответствовала набору документов. Постоянно поддерживать её в полностью актуальном состоянии обычно не требуется.

F.42.6. Ссылки

Сайт разработки GiST *http://www.sai.msu.su/~megera/postgres/gist/*

Сайт разработки Tsearch2 *http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/*

F.42.7. Авторы

Олег Бартунов `<oleg@sai.msu.su>`, Москва, Московский Государственный Университет, Россия

Фёдор Сигаев `<teodor@sigaev.ru>`, Москва, ООО «Дельта-Софт», Россия

Александр Коротков `<a.korotkov@postgrespro.ru>`, Москва, Postgres Professional, Россия

Документация: Кристофер Кингс-Линн

Разработку этого модуля спонсировало ООО «Дельта-Софт», г. Москва, Россия.

F.43. pg_tsparser

`pg_tsparser` — это расширение Postgres Pro для текстового поиска. Это расширение меняет стандартную стратегию разбора текста для слов, включающих:

- подчёркивания
- цифры и буквы, разделённые знаком минуса

В дополнение к отдельным частям слова, возвращаемых по умолчанию, `pg_tsparser` также возвращает всё слово.

F.43.1. Установка и настройка

Расширение `pg_tsparser` входит в состав Postgres Pro. Чтобы задействовать `pg_tsparser`, установив Postgres Pro, создайте расширение `pg_tsparser` в каждой базе данных, где вы планируете его использовать:

```
CREATE EXTENSION pg_tsparser;
```

Активировав `pg_tsparser`, вы можете создавать собственные конфигурации текстового поиска. Вместе с `pg_tsparser` можно использовать любой имеющийся словарь.

Например, так можно создать конфигурацию `english_ts` для английского языка:

```
CREATE TEXT SEARCH CONFIGURATION english_ts (
    PARSER = tsparser
);

COMMENT ON TEXT SEARCH CONFIGURATION english_ts IS 'text search configuration for
 english language';

ALTER TEXT SEARCH CONFIGURATION english_ts
    ADD MAPPING FOR email, file, float, host, hword_numpart, int,
    numhword, numword, sfloat, uint, url, url_path, version
    WITH simple;
```

```
ALTER TEXT SEARCH CONFIGURATION english_ts
    ADD MAPPING FOR asciiword, asciihword, hword_asciipart,
    word, hword, hword_part
    WITH english_stem;
```

F.43.2. Примеры

Следующие примеры иллюстрируют различия в результатах поиска, возвращаемых анализатором текста `pg_tsparser` и стандартным анализатором:

```
SELECT to_tsvector('english', 'pg_trgm') as def_parser,
       to_tsvector('english_ts', 'pg_trgm')  as new_parser;
   def_parser    |         new_parser
-----------------+----------------------------
 'pg':1 'trgm':2 | 'pg':2 'pg_trgm':1 'trgm':3
(1 row)

SELECT to_tsvector('english', '123-abc') as def_parser,
       to_tsvector('english_ts', '123-abc')  as new_parser;
   def_parser    |         new_parser
-----------------+----------------------------
 '123':1 'abc':2 | '123':2 '123-abc':1 'abc':3
(1 row)

SELECT to_tsvector('english', 'rel-3.2-A') as def_parser,
       to_tsvector('english_ts', 'rel-3.2-A')  as new_parser;
    def_parser    |          new_parser
------------------+------------------------------
 '-3.2':2 'rel':1 | '3.2':3 'rel':2 'rel-3.2-a':1
(1 row)
```

См. также

CREATE TEXT SEARCH CONFIGURATION

ALTER TEXT SEARCH CONFIGURATION

F.43.3. Авторы

Postgres Professional, Москва, Россия

F.44. pg_variables

Модуль `pg_variables` содержит функции для работы с переменными различных типов. Созданные переменные существуют в течение текущей пользовательской сессии.

F.44.1. Установка

Расширение `pg_variables` включено в состав Postgres Pro. Установив Postgres Pro, вы должны выполнить команду CREATE EXTENSION, чтобы подготовить `pg_variables` к работе, следующим образом:

```
CREATE EXTENSION pg_variables;
```

F.44.2. Использование

Модуль `pg_variables` содержит функции, позволяющие создавать переменные скалярных типов, переменные-записи и переменные-массивы, читать их и управлять ими. Синтаксис и описания функций представлены в следующих разделах:

- В Подразделе F.44.3.1 описываются функции для скалярных переменных.
- В Подразделе F.44.3.2 описываются функции для переменных-записей.
- В Подразделе F.44.3.3 описываются функции для переменных-массивов.
- В Подразделе F.44.3.4 перечисляются функции, позволяющие управлять всеми переменными в текущем сеансе.

Разобранные примеры использования вы можете найти в Подразделе F.44.4.

F.44.2.1. Использование транзакционных переменных

По умолчанию переменные создаются как нетранзакционные. Успешно созданная переменная продолжает существовать на протяжении всего сеанса, вне зависимости от возможных откатов транзакций. Например:

```
SELECT pgv_set('vars', 'int1', 101);
BEGIN;
SELECT pgv_set('vars', 'int2', 102);
ROLLBACK;

SELECT * FROM pgv_list() order by package, name;
 package | name | is_transactional
---------+------+------------------
 vars    | int1 | f
 vars    | int2 | f
```

Если вы хотите использовать переменную с поддержкой транзакций и точек сохранения, передайте дополнительный флаг `is_transactional` в последнем параметре функции, создающей переменную:

```
BEGIN;
SELECT pgv_set('vars', 'trans_int', 101, true);
SAVEPOINT sp1;
SELECT pgv_set('vars', 'trans_int', 102, true);
ROLLBACK TO sp1;
COMMIT;
SELECT pgv_get('vars', 'trans_int', NULL::int);
 pgv_get
---------
     101
```

Вы должны указывать флаг `is_transactional` при каждом изменении значения транзакционной переменной с помощью функций `pgv_set()` и `pgv_insert()`. В противном случае произойдёт ошибка. Другим функциям передавать этот флаг не нужно.

```
SELECT pgv_insert('pack', 'var_record', row(123::int, 'text'::text), true);

SELECT pgv_insert('pack', 'var_record', row(456::int, 'another text'::text));
ERROR:  variable "var_record" already created as TRANSACTIONAL

SELECT pgv_delete('pack', 'var_record', 123::int);
```

Если вызовы функций `pgv_free()` или `pgv_remove()` откатываются, затрагиваемые транзакционные переменные восстанавливаются, в отличие от нетранзакционных, которые удаляются необратимо. Например:

```
SELECT pgv_set('pack', 'var_reg', 123);
SELECT pgv_set('pack', 'var_trans', 456, true);
BEGIN;
SELECT pgv_free();
ROLLBACK;
```

```
SELECT * FROM pgv_list();
 package |   name    | is_transactional
---------+-----------+------------------
 pack    | var_trans | t
```

F.44.3. Функции

F.44.3.1. Скалярные переменные

Следующие функции поддерживают скалярные переменные:

Функция	Возвращает
pgv_set(package text, name text, value anynonarray, is_transactional bool default false)	void
pgv_get(package text, name text, var_type anynonarray, strict bool default true)	anynonarray

Для функции pgv_get() вы должны сначала создать пакет и переменную, воспользовавшись функцией pgv_set(). Если указанный пакет или переменная не существуют, происходит ошибка:

```
SELECT pgv_get('vars', 'int1', NULL::int);
ERROR:  unrecognized package "vars"

SELECT pgv_get('vars', 'int1', NULL::int);
ERROR:  unrecognized variable "int1"
```

Функция pgv_get() проверяет тип переменной. Если заданный тип не соответствует типу переменной, выдаётся ошибка:

```
SELECT pgv_get('vars', 'int1', NULL::text);
ERROR:  variable "int1" requires "integer" value
```

F.44.3.2. Записи

Следующие функции поддерживают наборы переменных типа запись.

Функция	Возвращает	Описание
pgv_insert(package text, name text, r record, is_transactional bool default false)	void	Вставляет запись в набор переменных для заданного пакета. Если пакет или переменная не существуют, они создаются автоматически. Первый столбец записи r — первичный ключ. Если запись с таким же первичным ключом уже существует или этот набор переменных имеет другую структуру, выдаётся ошибка.
pgv_update(package text, name text, r record)	boolean	Изменяет запись с соответствующим первичным ключом (он задаётся в первом столбце r). Возвращает true, если запись была найдена. Если этот набор переменных

Функция	Возвращает	Описание
		имеет другую структуру, выдаётся ошибка.
pgv_delete(package text, name text, value anynonarray)	boolean	Удаляет запись с соответствующим первичным ключом (он задаётся в первом столбце r). Возвращает true, если запись была найдена.
pgv_select(package text, name text)	set of records	Возвращает записи из набора переменных.
pgv_select(package text, name text, value anynonarray)	record	Возвращает записи из набора переменных с соответствующими первичными ключами (первичный ключ задаётся в первом столбце r).
pgv_select(package text, name text, value anyarray)	set of records	Возвращает записи из набора переменных с соответствующими первичными ключами (первичный ключ задаётся в первом столбце r).

Для использования функций pgv_update(), pgv_delete() и pgv_select() вы должны сначала создать пакет и переменную, воспользовавшись функцией pgv_insert(). Тип переменной и тип записи должны быть одинаковыми, иначе выдаётся ошибка.

F.44.3.3. Массивы

Следующие функции поддерживают переменные-массивы:

Функция	Возвращает
pgv_set(package text, name text, value anyarray, is_transactional bool default false)	void
pgv_get(package text, name text, var_type anyarray, strict bool default true)	anyarray

Инструкции по использованию этих функций совпадают с приведёнными в Подразделе F.44.3.1 для скалярных переменных.

F.44.3.4. Функции разного назначения

Функция	Возвращает	Описание
pgv_exists(package text, name text)	bool	Возвращает true, если существует пакет и переменная.
pgv_exists(package text)	bool	Возвращает true, если существует указанный пакет.
pgv_remove(package text, name text)	void	Удаляет переменную с заданным именем. Указанный пакет и переменная должны существовать, иначе выдаётся ошибка.

Функция	Возвращает	Описание
`pgv_remove(package text)`	`void`	Удаляет заданный пакет и все его переменные. Указанный пакет должен существовать, иначе выдаётся ошибка.
`pgv_free()`	`void`	Удаляет все пакеты и переменные.
`pgv_list()`	`table(package text, name text, is_transactional bool)`	Выводит список всех существующих переменных с именем соответствующего пакета и признаком того, что переменная транзакционная.
`pgv_stats()`	`table(package text, allocated_memory bigint)`	Возвращает список созданных пакетов и объём памяти, используемый переменными, в байтах. Если вы используете транзакционные переменные, в этот список также включаются все удалённые пакеты, которые могут быть восстановлены командой `ROLLBACK`. Эта функция поддерживается только в Postgres Pro версии 9.6 и новее.

F.44.3.5. Устаревшие функции

F.44.3.5.1. Целочисленные переменные

Следующие функции считаются устаревшими. Используйте вместо них универсальные функции для скалярных переменных.

Функция	Возвращает
`pgv_set_int(package text, name text, value int, is_transactional bool default false)`	`void`
`pgv_get_int(package text, name text, strict bool default true)`	`int`

F.44.3.5.2. Текстовые переменные

Следующие функции считаются устаревшими. Используйте вместо них универсальные функции для скалярных переменных.

Функция	Возвращает
`pgv_set_text(package text, name text, value text, is_transactional bool default false)`	`void`
`pgv_get_text(package text, name text, strict bool default true)`	`text`

F.44.3.5.3. Числовые переменные

Следующие функции считаются устаревшими. Используйте вместо них универсальные функции для скалярных переменных.

Функция	Возвращает
pgv_set_numeric(package text, name text, value numeric, is_transactional bool default false)	void
pgv_get_numeric(package text, name text, strict bool default true)	numeric

F.44.3.5.4. Переменные даты/времени

Следующие функции считаются устаревшими. Используйте вместо них универсальные функции для скалярных переменных.

Функция	Возвращает
pgv_set_timestamp(package text, name text, value timestamp, is_transactional bool default false)	void
pgv_get_timestamp(package text, name text, strict bool default true)	timestamp

F.44.3.5.5. Переменные даты/времени с часовым поясом

Следующие функции считаются устаревшими. Используйте вместо них универсальные функции для скалярных переменных.

Функция	Возвращает
pgv_set_timestamptz(package text, name text, value timestamptz, is_transactional bool default false)	void
pgv_get_timestamptz(package text, name text, strict bool default true)	timestamptz

F.44.3.5.6. Переменные даты

Следующие функции считаются устаревшими. Используйте вместо них универсальные функции для скалярных переменных.

Функция	Возвращает
pgv_set_date(package text, name text, value date, is_transactional bool default false)	void
pgv_get_date(package text, name text, strict bool default true)	date

F.44.3.5.7. Переменные Jsonb

Следующие функции считаются устаревшими. Используйте вместо них универсальные функции для скалярных переменных.

Функция	Возвращает
pgv_set_jsonb(package text, name text, value jsonb, is_transactional bool default false)	void
pgv_get_jsonb(package text, name text, strict bool default true)	jsonb

F.44.4. Примеры

Определение скалярных переменных с помощью функции `pgv_set()` и получение их значений с помощью `pgv_get()`:

```
SELECT pgv_set('vars', 'int1', 101);
SELECT pgv_set('vars', 'int2', 102);
SELECT pgv_set('vars', 'text1', 'text variable'::text);

SELECT pgv_get('vars', 'int1', NULL::int);
 pgv_get
-------------
         101

SELECT pgv_get('vars', 'int2', NULL::int);
 pgv_get
-------------
         102

SELECT pgv_get('vars', 'text1', NULL::text);
 pgv_get
---------------
 text variable
```

В предположении, что создана следующая таблица `tab`, рассмотрите несколько примеров с переменными-записями:

```
CREATE TABLE tab (id int, t varchar);
INSERT INTO tab VALUES (0, 'str00'), (1, 'str11');
```

Для работы с переменными-записями можно использовать следующие функции:

```
SELECT pgv_insert('vars', 'r1', tab) FROM tab;

SELECT pgv_select('vars', 'r1');
 pgv_select
------------
 (1,str11)
 (0,str00)

SELECT pgv_select('vars', 'r1', 1);
 pgv_select
------------
 (1,str11)

SELECT pgv_select('vars', 'r1', 0);
 pgv_select
------------
 (0,str00)

SELECT pgv_select('vars', 'r1', ARRAY[1, 0]);
 pgv_select
------------
 (1,str11)
 (0,str00)

SELECT pgv_delete('vars', 'r1', 1);

SELECT pgv_select('vars', 'r1');
 pgv_select
```

```
------------
 (0,str00)
```

Проанализируйте поведение транзакционной переменной `var_text` при изменении до и после точки сохранения:

```
SELECT pgv_set('pack', 'var_text', 'before transaction block'::text, true);
BEGIN;
SELECT pgv_set('pack', 'var_text', 'before savepoint'::text, true);
SAVEPOINT sp1;
SELECT pgv_set('pack', 'var_text', 'savepoint sp1'::text, true);
SAVEPOINT sp2;
SELECT pgv_set('pack', 'var_text', 'savepoint sp2'::text, true);
RELEASE sp2;
SELECT pgv_get('pack', 'var_text', NULL::text);
    pgv_get
---------------
 savepoint sp2

ROLLBACK TO sp1;
SELECT pgv_get('pack', 'var_text', NULL::text);
     pgv_get
------------------
 before savepoint

ROLLBACK;
SELECT pgv_get('pack', 'var_text', NULL::text);
         pgv_get
--------------------------
 before transaction block
```

Если вы создадите транзакционную переменную после команд `BEGIN` или `SAVEPOINT` и затем произойдёт откат к предыдущему состоянию, эта переменная будет удалена:

```
BEGIN;
SAVEPOINT sp1;
SAVEPOINT sp2;
SELECT pgv_set('pack', 'var_int', 122, true);
RELEASE SAVEPOINT sp2;
SELECT pgv_get('pack', 'var_int', NULL::int);
pgv_get
---------
     122

ROLLBACK TO sp1;
SELECT pgv_get('pack','var_int', NULL::int);
ERROR:  unrecognized variable "var_int"
COMMIT;
```

Просмотреть имеющиеся пакеты и переменные:

```
SELECT * FROM pgv_list() ORDER BY package, name;
 package |   name   | is_transactional
---------+----------+------------------
 pack    | var_text | t
 vars    | int1     | f
 vars    | int2     | f
 vars    | r1       | f
 vars    | text1    | f
```

Получение объёма памяти, занятой переменными, в байтах:

```
SELECT * FROM pgv_stats() ORDER BY package;
 package | allocated_memory
---------+------------------
 pack    |            16384
 vars    |            32768
```

Удаление избранных переменных или пакетов:

```
SELECT pgv_remove('vars', 'int1');
SELECT pgv_remove('vars');
```

Удаление всех пакетов и переменных:

```
SELECT pgv_free();
```

F.44.5. Авторы

Postgres Professional, Москва, Россия

F.45. pg_visibility

Модуль `pg_visibility` даёт возможность исследовать для определённой таблицы карту видимости (Visibility Map, VM) и информацию о видимости на уровне страниц. Он также предоставляет функции для проверки целостности карты видимости и принудительного её пересоздания.

Для хранения информации о видимости на уровне страниц применяются по три различных бита. Бит полной видимости в карте показывает, что каждый кортеж в соответствующей странице отношения является видимым для всех текущих и будущих транзакций. Бит полной заморозки в карте видимости показывает, что все кортежи в данной странице являются замороженными; то есть никакой операции очистки в будущем не придётся обрабатывать эту страницу, пока в ней не будет добавлен, изменён, удалён или заблокирован кортеж. Бит `PD_ALL_VISIBLE` в заголовке страницы имеет то же значение, что и бит полной видимости в карте видимости, но он хранится в самой странице данных, а не в отдельной структуре данных. В обычной ситуации эти два бита будут согласованы, но бит полной видимости в странице иногда может быть установлен, тогда как в карте видимости он оказывается сброшенным при восстановлении после сбоя. Считываемые значения могут также различаться, если они подвергаются изменению в промежутке между обращениями `pg_visibility` к карте видимости и к странице данных. Разумеется, эти биты также могут различаться при событиях, приводящих к разрушению данных.

Функции, выдающие информацию о битах `PD_ALL_VISIBLE`, более дорогостоящие, чем те, что обращаются только к карте видимости, так как они должны читать блоки отношения, а не только карту видимости (которая намного меньше). Дорогостоящими являются также и функции, проверяющие блоки данных отношения.

F.45.1. Функции

`pg_visibility_map(relation regclass, blkno bigint, all_visible OUT boolean, all_frozen OUT boolean) returns record`

Возвращает биты полной видимости и полной заморозки в карте видимости для указанного блока заданного отношения.

`pg_visibility(relation regclass, blkno bigint, all_visible OUT boolean, all_frozen OUT boolean, pd_all_visible OUT boolean) returns record`

Возвращает биты полной видимости и полной заморозки в карте видимости для указанного блока заданного отношения, а также бит `PD_ALL_VISIBLE` этого блока.

`pg_visibility_map(relation regclass, blkno OUT bigint, all_visible OUT boolean, all_frozen OUT boolean) returns setof record`

Возвращает биты полной видимости и полной заморозки в карте видимости для каждого блока заданного отношения.

`pg_visibility(relation regclass, blkno OUT bigint, all_visible OUT boolean, all_frozen OUT boolean, pd_all_visible OUT boolean) returns setof record`

Возвращает биты полной видимости и полной заморозки в карте видимости для каждого блока заданного отношения, а также бит `PD_ALL_VISIBLE` каждого блока.

`pg_visibility_map_summary(relation regclass, all_visible OUT bigint, all_frozen OUT bigint) returns record`

Возвращает число полностью видимых страниц и полностью замороженных страниц в отношении, согласно карте видимости.

`pg_check_frozen(relation regclass, t_ctid OUT tid) returns setof tid`

Возвращает идентификаторы TID незамороженных кортежей в страницах, помеченных как полностью замороженные в карте видимости. Если эта функция возвращает непустой набор TID, карта видимости испорчена.

`pg_check_visible(relation regclass, t_ctid OUT tid) returns setof tid`

Возвращает идентификаторы TID не полностью видимых кортежей в страницах, помеченных как полностью видимые в карте видимости. Если эта функция возвращает непустой набор TID, карта видимости испорчена.

`pg_truncate_visibility_map(relation regclass) returns void`

Аннулирует карту видимости для заданного отношения. Эта функция полезна, если вы считаете, что карта видимости для указанного отношения испорчена, и хотите принудительно пересоздать её. Первая же команда `VACUUM`, выполняемая с данным отношением после этой функции, просканирует все страницы в отношении и пересоздаст карту видимости. (Пока это не произойдёт, для запросов карта видимости будет выглядеть как полностью нулевая.)

По умолчанию эти функции разрешено выполнять только суперпользователям и членам роли `pg_stat_scan_tables`, за исключением `pg_truncate_visibility_map(relation regclass)`, которую могут выполнять только суперпользователи.

F.45.2. Автор

Роберт Хаас `<rhaas@postgresql.org>`

F.46. plantuner

Модуль `plantuner` добавляет поддержку указаний для планировщика, позволяющих отключать или подключать определённые индексы при выполнении запроса.

F.46.1. Объяснение

В некоторых случаях может потребоваться управлять планировщиком, давая ему указания, чтобы оптимизатор не выполнял некоторые части своего алгоритма. В частности, нередко возникают ситуации, когда разработчик хочет временно отключить некоторые индексы, не удаляя их, либо наоборот, использовать определённый индекс принудительно.

Эта версия `plantuner` даёт возможность скрыть определённые индексы от планировщика Postgres Pro, чтобы он их не использовал. С некоторой нагрузкой Postgres Pro бывает слишком пессимистичным в отношении только что созданных таблиц и считает, что в них содержится гораздо больше строк, чем есть на самом деле. Если же переменная GUC `plantuner.fix_empty_table` имеет значение `true`, `plantuner` будет обнулять число страниц/кортежей в таблице, которая не содержит никаких блоков в файле.

F.46.2. Переменные GUC

`plantuner.disable_index` — список индексов, которые не будет видеть планировщик.

plantuner.enable_index — список индексов, которые будет видеть планировщик, даже если они скрыты параметром plantuner.disable_index.

F.46.3. Пример

Чтобы включить этот модуль, вы можете либо загрузить разделяемую библиотеку plantuner в сеансе psql, либо указать её в параметре shared_preload_libraries в postgresql.conf.

```
=# LOAD 'plantuner';
=# create table test(id int);
=# create index id_idx on test(id);
=# create index id_idx2 on test(id);
=# \d test
     Table "public.test"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer |
Indexes:
    "id_idx" btree (id)
    "id_idx2" btree (id)
=# explain select id from test where id=1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=4.34..15.03 rows=12 width=4)
   Recheck Cond: (id = 1)
   ->  Bitmap Index Scan on id_idx2  (cost=0.00..4.34 rows=12 width=0)
         Index Cond: (id = 1)
(4 rows)
=# set enable_seqscan=off;
=# set plantuner.disable_index='id_idx2';
=# explain select id from test where id=1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=4.34..15.03 rows=12 width=4)
   Recheck Cond: (id = 1)
   ->  Bitmap Index Scan on id_idx  (cost=0.00..4.34 rows=12 width=0)
         Index Cond: (id = 1)
(4 rows)
=# set plantuner.disable_index='id_idx2,id_idx';
=# explain select id from test where id=1;
                               QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on test  (cost=10000000000.00..10000000040.00 rows=12 width=4)
   Filter: (id = 1)
(2 rows)
=# set plantuner.enable_index='id_idx';
=# explain select id from test where id=1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=4.34..15.03 rows=12 width=4)
   Recheck Cond: (id = 1)
   ->  Bitmap Index Scan on id_idx  (cost=0.00..4.34 rows=12 width=0)
         Index Cond: (id = 1)
(4 rows)
```

F.46.4. Авторы

Разработку осуществили Фёдор Сигаев (teodor@sigaev.ru) и Олег Бартунов (oleg@sai.msu.su).

Эту работу спонсировал проект Nomao (http://www.nomao.com).

F.47. postgres_fdw

Модуль `postgres_fdw` предоставляет обёртку сторонних данных `postgres_fdw`, используя которую можно обращаться к данным, находящимся на внешних серверах Postgres Pro.

Функциональность этого модуля во многом пересекается с функциональностью старого модуля dblink. Однако `postgres_fdw` предоставляет более прозрачный и стандартизированный синтаксис для обращения к удалённым таблицам и во многих случаях даёт лучшую производительность.

Чтобы подготовиться к обращению к удалённым данным через `postgres_fdw`:

1. Установите расширение `postgres_fdw` с помощью команды CREATE EXTENSION.
2. Создайте объект стороннего сервера, используя CREATE SERVER, который будет представлять удалённую базу данных, к которой вы хотите подключаться. Укажите свойства подключения, кроме `user` и `password`, в параметрах объекта сервера.
3. Создайте сопоставление пользователей, используя CREATE USER MAPPING, для каждого пользователя базы, которому нужен доступ к удалённому серверу. Укажите имя и пароль удалённого пользователя в параметрах `user` и `password` сопоставления.
4. Создайте стороннюю таблицу, используя CREATE FOREIGN TABLE или IMPORT FOREIGN SCHEMA, для каждой удалённой таблицы, к которой вы хотите обращаться. Столбцы сторонней таблицы должны соответствовать столбцам целевой удалённой таблицы. Однако вы можете использовать локально имена таблиц и/или столбцов, отличные от удалённых, если укажете корректные удалённые имена в параметрах объекта сторонней таблицы.

После этого для обращения к данным, хранящимся в нижележащей удалённой таблице, вам нужно только выполнять `SELECT`. Вы также можете изменять данные в удалённой таблице, выполняя `INSERT`, `UPDATE` или `DELETE`. (Разумеется, удалённый пользователь, указанный в сопоставлении, должен иметь необходимые права для этого.)

Заметьте, что в настоящее время в `postgres_fdw` не поддерживаются операторы `INSERT` с предложением `ON CONFLICT DO UPDATE`. Однако предложение `ON CONFLICT DO NOTHING` поддерживается, при отсутствии указания для выбора уникального индекса.

Обычно рекомендуется объявлять столбцы сторонней таблицы точно с теми же типами данных и правилами сортировки, если они применимы, как у целевых столбцов удалённой таблицы. Хотя `postgres_fdw` в настоящее время довольно лоялен к преобразованиям типов данных при необходимости, но когда типы или правила сортировки не совпадают, возможны неожиданные семантические аномалии, вследствие того, что удалённый сервер будет обрабатывать условия запроса не так, как локальный сервер.

Заметьте, что сторонняя таблица может быть объявлена с меньшим количеством или с другим порядком столбцов, чем в нижележащей удалённой таблице. Сопоставление столбцов удалённой таблицы осуществляется по имени, а не по позиции.

F.47.1. Параметры обёртки для postgres_fdw

F.47.1.1. Параметры подключения

Для стороннего сервера, настраиваемого с использованием обёртки сторонних данных `postgres_fdw`, можно задать те же параметры, что принимает libpq в строках подключения, как описано в Подразделе 31.1.2, за исключением следующих недопустимых параметров:

- `user` и `password` (их следует задавать в сопоставлениях пользователей)
- `client_encoding` (автоматически принимается равной локальной кодировке сервера)
- `fallback_application_name` (всегда `postgres_fdw`)

Подключаться к сторонним серверам без аутентификации по паролю могут только суперпользователи, поэтому в сопоставлениях для обычных пользователей всегда нужно задавать пароль (`password`).

F.47.1.2. Параметры имени объекта

Эти параметры позволяют управлять тем, как на удалённый сервер Postgres Pro будут передаваться имена, фигурирующие в операторах SQL. Данные параметры нужны, когда сторонняя таблица создаётся с именами, отличными от имён удалённой таблицы.

`schema_name`

Этот параметр, который может задаваться для сторонней таблицы, указывает имя схемы для обращения к этой таблице на удалённом сервере. Если данный параметр опускается, применяется схема сторонней таблицы.

`table_name`

Этот параметр, который может задаваться для сторонней таблицы, указывает имя таблицы для обращения к этой таблице на удалённом сервере. Если данный параметр опускается, применяется имя сторонней таблицы.

`column_name`

Этот параметр, который может задаваться для столбца сторонней таблицы, указывает имя столбца для обращения к этому столбцу на удалённом сервере. Если данный параметр опускается, применяется исходное имя столбца.

F.47.1.3. Параметры оценки стоимости

Модуль `postgres_fdw` получает удалённые данные, выполняя запросы на удалённых серверах, поэтому в идеале ожидаемая стоимость сканирования сторонней таблицы должна равняться стоимости выполнения на удалённом сервере плюс издержки сетевого взаимодействия. Самый надёжный способ получить такие оценки — узнать стоимость у удалённого сервера и добавить некоторую надбавку — но для простых запросов может быть невыгодно передавать дополнительный запрос, только чтобы получить оценку стоимости. Поэтому `postgres_fdw` предоставляет следующие параметры, позволяющие управлять вычислением оценки стоимости:

`use_remote_estimate`

Этот параметр, который может задаваться для сторонней таблицы или для стороннего сервера, определяет, будет ли `postgres_fdw` выполнять удалённо команды `EXPLAIN` для получения оценок стоимости. Параметр, заданный для сторонней таблицы, переопределяет параметр сервера, но только для данной таблицы. Значение по умолчанию — `false` (выкл.).

`fdw_startup_cost`

Этот параметр, который может задаваться для стороннего сервера, устанавливает числовое значение, добавляемое к оценке стоимости запуска для любого сканирования сторонней таблицы на этом сервере. Он выражает дополнительные издержки на установление подключения, разбор и планирование запроса на удалённой стороне и т. д. Значение по умолчанию — `100`.

`fdw_tuple_cost`

Этот параметр, который может задаваться для стороннего сервера, устанавливает числовое значение, выражающее дополнительную цену чтения одного кортежа из сторонней таблицы на этом сервере. Это число можно увеличить или уменьшить, отражая меньшую или большую фактическую скорость сетевого взаимодействия с удалённым сервером. Значение по умолчанию — `0.01`.

Когда поведение `use_remote_estimate` включено, `postgres_fdw` получает количество строк и оценку стоимости с удалённого сервера, а затем добавляет к оценке стоимости `fdw_startup_cost` и `fdw_tuple_cost`. Когда поведение `use_remote_estimate` отключено, `postgres_fdw` рассчитывает число строк и оценку стоимости локально, а затем так же добавляет к этой оценке `fdw_startup_cost` и `fdw_tuple_cost`. Локальная оценка может быть точной только при условии наличия локальной копии статистики удалённых таблиц. Обновить эту статистику для сторонней

таблицы можно с помощью команды ANALYZE; при этом удалённая таблица будет просканирована, и по её содержимому будут вычислена и сохранена статистика как для локальной таблицы. Локальное хранение статистики может быть полезно для сокращения издержек планирования для удалённой таблицы — но если удалённая таблица меняется часто, локальная статистика будет быстро устаревать.

F.47.1.4. Параметры удалённого выполнения

По умолчанию ограничения `WHERE`, содержащие встроенные операторы и функции, обрабатываются на удалённом сервере, а ограничения, содержащие вызовы не встроенных функций, проверяются локально после получения строк. Если же расширенные функции доступны на удалённом сервере и можно рассчитывать, что они дадут те же результаты, что и локально, производительность можно увеличить, передавая и такие предложения `WHERE` для удалённого выполнения. Этим поведением позволяет управлять следующий параметр:

`extensions`

В этом параметре задаётся список имён расширений Postgres Pro через запятую, которые установлены и имеют совместимые версии и на локальном, и на удалённом сервере. Относящиеся к перечисленным расширениям и при этом постоянные (immutable) функции и операторы могут передаваться на выполнение удалённому серверу. Этот параметр можно задать только для стороннего сервера, но не для таблицы.

При использовании параметра `extensions` *пользователь сам отвечает* за то, чтобы перечисленные расширения существовали и их поведение было одинаковым на локальном и удалённом сервере. В противном случае удалённые запросы могут выдавать ошибки или неожиданные результаты.

`fetch_size`

Этот параметр определяет, сколько строк должна получать `postgres_fdw` в одной операции выборки. Его можно задать для сторонней таблицы или стороннего сервера. Значение по умолчанию — `100` строк.

F.47.1.5. Параметры изменения данных

По умолчанию все сторонние таблицы, доступные через `postgres_fdw`, считаются допускающими изменения. Это можно переопределить с помощью следующего параметра:

`updatable`

Этот параметр определяет, будет ли `postgres_fdw` допускать изменения в сторонних таблицах посредством команд `INSERT`, `UPDATE` и `DELETE`. Его можно задать для сторонней таблицы или для стороннего сервера. Параметр, определённый на уровне таблицы, переопределяет параметр уровня сервера. Значение по умолчанию — `true` (изменения разрешены).

Конечно, если удалённая таблица на самом деле не допускает изменения, всё равно произойдёт ошибка. Использование этого параметра прежде всего позволяет выдать ошибку локально, не обращаясь к удалённому серверу. Заметьте, однако, что представление `information_schema` будет показывать, что определённая сторонняя таблица `postgres_fdw` является изменяемой (или нет), согласно значению данного параметра, не проверяя это на удалённом сервере.

F.47.1.6. Параметры импорта

Обёртка `postgres_fdw` позволяет импортировать определения сторонних таблиц с применением команды IMPORT FOREIGN SCHEMA. Эта команда создаёт на локальном сервере определения сторонних таблиц, соответствующие таблицам или представлениям, существующим на удалённом сервере. Если импортируемые удалённые таблицы содержат столбцы пользовательских типов данных, на локальном сервере должны быть совместимые типы с теми же именами.

Поведение процедуры импорта можно настроить следующими параметрами (задаваемыми в команде `IMPORT FOREIGN SCHEMA`):

import_collate

Этот параметр устанавливает, будут ли в определения сторонних таблиц, импортируемых с внешнего сервера, включаться характеристики столбцов COLLATE. По умолчанию они включаются. Вам может потребоваться отключить его, если на удалённом сервере набор имён правил сортировки отличается от локального, что скорее всего будет иметь место, если серверы работают в разных операционных системах. Тем не менее отключение чревато тем, что правила сортировки в столбцах импортируемых таблиц не совпадут с правилами для нижележащих данных, что приведёт к аномальному поведению запросов.

Даже когда этот параметр равен true, импортировать столбцы с правилом сортировки, выбираемым на удалённом сервере по умолчанию, рискованно. Столбцы импортируются со свойством COLLATE "default", а в результате будет применяться правило сортировки, выбираемое по умолчанию на локальном сервере, которое может отличаться от выбираемого на удалённом.

import_default

Этот параметр устанавливает, будут ли в определения сторонних таблиц, импортируемых с внешнего сервера, включаться заданные для столбцов выражения DEFAULT. По умолчанию они не включаются. Если вы включите этот параметр, остерегайтесь выражений по умолчанию, которые могут вычисляться на локальном сервере не так, как на удалённом; например, частый источник проблем — nextval(). Если в импортируемом выражении используются функции или операторы, несуществующие локально, команда IMPORT в целом выдаст сбой.

import_not_null

Этот параметр устанавливает, будут ли в определения сторонних таблиц, импортируемых с внешнего сервера, включаться ограничения столбцов NOT NULL. По умолчанию они включаются.

Заметьте, что никакие другие ограничения, кроме NOT NULL, из удалённых таблиц импортироваться не будут. Хотя Postgres Pro поддерживает ограничения CHECK для сторонних таблиц, никаких средств для автоматического импорта их нет из-за риска различного вычисления выражения ограничения на локальном и удалённом серверах. Любая такая несогласованность в поведении ограничений CHECK могла бы привести к сложно выявляемым ошибкам в оптимизации запросов. Поэтому, если вы хотите импортировать ограничения CHECK, вы должны сделать это вручную и должны внимательно проверить семантику каждого. Более подробно интерпретация ограничений CHECK для сторонних таблиц описана в CREATE FOREIGN TABLE.

Таблицы или сторонние таблицы, являющиеся секциями некоторой другой таблицы, исключаются автоматически. Секционированные таблицы импортируются, только если они не являются секциями каких-либо других таблиц. Так как все данные могут быть доступны через секционированную таблицу, являющуюся вершиной в иерархии секционирования, при таком подходе должен быть возможен доступ ко всем данным без создания дополнительных объектов.

F.47.2. Управление соединением

Модуль postgres_fdw устанавливает соединение со сторонним сервером при первом запросе, в котором участвует сторонняя таблица, связанная со сторонним сервером. Это соединение сохраняется и повторно используется для последующих запросов в том же сеансе. Однако если для обращения к стороннему серверу задействуются разные пользователи (сопоставления пользователей), отдельное соединение устанавливается для каждого сопоставления пользователей.

F.47.3. Управление транзакциями

В процессе выполнения запроса, в котором участвуют какие-либо удалённые таблицы на стороннем сервере, postgres_fdw открывает транзакцию на удалённом сервере, если такая транзакция ещё не была открыта для текущей локальной транзакции. Эта удалённая транзакция

фиксируется или прерывается, когда фиксируется или прерывается локальная транзакция. Подобным образом реализуется и управление точками сохранения.

Для удалённой транзакции выбирается режим изоляции SERIALIZABLE, когда локальная транзакция открыта в режиме SERIALIZABLE; в противном случае применяется режим REPEATABLE READ. Этот выбор гарантирует, что если запрос сканирует несколько таблиц на удалённом сервере, он будет получать согласованные данные одного снимка для всех сканирований. Как следствие, последовательные запросы в одной транзакции будут видеть одни данные удалённого сервера, даже если на нём параллельно происходят изменения, вызванные другими действиями. Это поведение ожидаемо для локальной транзакции в режимах SERIALIZABLE и REPEATABLE READ, но для локальной транзакции в режиме READ COMMITTED оно может быть неожиданным. В будущих выпусках Postgres Pro эти правила могут быть изменены.

Заметьте, что подготовку удалённой транзакции для двухфазной фиксации postgres_fdw в настоящее время не поддерживает.

F.47.4. Оптимизация удалённых запросов

Обёртка postgres_fdw пытается оптимизировать удалённые запросы, уменьшая объём обмена данными со сторонними серверами. Для этого она может передавать на выполнение удалённому серверу предложения WHERE и не получать столбцы таблицы, не требующиеся для текущего запроса. Чтобы уменьшить риск некорректного выполнения запросов, предложения WHERE передаются удалённому серверу, только если в них используются типы данных, операторы и функции, встроенные в ядро или относящиеся к расширениям, перечисленным в параметре extensions. Операторы и функции в таких предложениях также должны быть постоянными (IMMUTABLE). Для запросов UPDATE или DELETE обёртка postgres_fdw пытается оптимизировать выполнение, передавая весь запрос на удалённый сервер, если в запросе нет предложения WHERE, которое нельзя было бы передать, не выполняется локальное соединение, в целевой таблице отсутствуют локальные триггеры BEFORE/AFTER уровня строки и в родительских представлениях нет ограничения CHECK OPTION. Кроме того, в запросах UPDATE выражения, присваиваемые целевым столбцам, должны задействовать только встроенные типы данных и постоянные (IMMUTABLE) операторы и функции, чтобы уменьшить риск неверного выполнения запроса.

Когда обёртка postgres_fdw обнаруживает соединение сторонних таблиц на одном стороннем сервере, она передаёт всё соединение этому серверу, если только по какой-то причине не решит, что будет эффективнее выбирать строки из каждой таблицы по отдельности, или сопоставляемые при обращении к таблицам пользователи оказываются разными. При передаче предложений JOIN принимаются те же меры предосторожности, что были описаны выше для предложений WHERE.

Запрос, фактически отправляемый удалённому серверу для выполнения, можно изучить с помощью команды EXPLAIN VERBOSE.

F.47.5. Окружение удалённого выполнения запросов

В удалённых сеансах, установленных обёрткой postgres_fdw, в параметре search_path задаётся только pg_catalog, так что без указания схемы видны только встроенные объекты. Это не проблема для запросов, которые генерирует сама postgres_fdw, так как она всегда добавляет такие указания. Однако это может быть опасно для функций, которые выполняются на удалённом сервере при срабатывании триггеров или правил для удалённых таблиц. Например, если удалённая таблица на самом деле представляет собой представление, любые функции, используемые в этом представлении, будут выполняться с таким ограниченным путём поиска. Поэтому рекомендуется в таких функциях дополнять схемой все имена либо добавлять параметры SET search_path (см. CREATE FUNCTION) в определения таких функций, чтобы установить ожидаемый ими путь поиска в окружении.

Обёртка postgres_fdw подобным образом устанавливает для удалённого сеанса различные параметры:

- TimeZone — UTC

- DateStyle — `ISO`
- IntervalStyle — `postgres`
- extra_float_digits принимает значение `3` для удалённых серверов версии 9.0 и новее либо `2` для более старых версий

С ними проблемы менее вероятны, чем с `search_path`, но если они возникнут, их можно урегулировать, установив нужные значения с помощью `SET`.

Это поведение *не* рекомендуется переопределять, устанавливая значения этих параметров на уровне сеанса; это скорее всего приведёт к поломке `postgres_fdw`.

F.47.6. Совместимость с разными версиями

Модуль `postgres_fdw` может применяться с удалёнными серверами версий, начиная с PostgreSQL 8.3. Способность только чтения данных доступна, начиная с 8.1. Однако при этом есть ограничение, вызванное тем, что `postgres_fdw` полагает, что постоянные встроенные функции и операторы могут безопасно передаваться на удалённый сервер для выполнения, если они фигурируют в предложении `WHERE` для сторонней таблицы. Таким образом, встроенная функция, добавленная в более новой версии, чем на удалённом сервере, может быть отправлена на выполнение, что в результате приведёт к ошибке «функция не существует» или подобной. Отказы такого типа можно предотвратить, переписав запрос, например, поместив ссылку на стороннюю таблицу во вложенный `SELECT` с `OFFSET 0` в качестве защиты от оптимизации, и применив проблематичную функцию или оператор снаружи этого вложенного `SELECT`.

F.47.7. Примеры

Ниже приведён пример создания сторонней таблицы с применением `postgres_fdw`. Сначала установите расширение:

```
CREATE EXTENSION postgres_fdw;
```

Затем создайте сторонний сервер с помощью команды CREATE SERVER. В данном примере мы хотим подключиться к серверу Postgres Pro, работающему по адресу `192.83.123.89`, порт `5432`. База данных, к которой устанавливается подключение, на удалённом сервере называется `foreign_db`:

```
CREATE SERVER foreign_server
        FOREIGN DATA WRAPPER postgres_fdw
        OPTIONS (host '192.83.123.89', port '5432', dbname 'foreign_db');
```

Для определения роли, которая будет задействована на удалённом сервере, с помощью CREATE USER MAPPING задаётся сопоставление пользователей:

```
CREATE USER MAPPING FOR local_user
        SERVER foreign_server
        OPTIONS (user 'foreign_user', password 'password');
```

Теперь можно создать стороннюю таблицу, применив команду CREATE FOREIGN TABLE. В этом примере мы хотим обратиться к таблице `some_schema.some_table` на удалённом сервере. Локальным именем этой таблицы будет `foreign_table`:

```
CREATE FOREIGN TABLE foreign_table (
        id integer NOT NULL,
        data text
)
        SERVER foreign_server
        OPTIONS (schema_name 'some_schema', table_name 'some_table');
```

Важно, чтобы типы данных и другие свойства столбцов, объявленных в `CREATE FOREIGN TABLE`, соответствовали фактической удалённой таблице. Также должны соответствовать имена столбцов, если только вы не добавите параметры `column_name` для отдельных столбцов, задающие их

реальные имена в удалённой таблице. Во многих случаях использовать IMPORT FOREIGN SCHEMA предпочтительнее, чем конструировать определения сторонних таблиц вручную.

F.47.8. Автор

Шигеру Ханада <shigeru.hanada@gmail.com>

F.48. seg

Этот модуль реализует тип данных `seg` для представления отрезков или интервалов чисел с плавающей точкой. Тип `seg` может выражать отсутствие уверенности в границах интервала, что позволяет применять его для представления лабораторных измерений.

F.48.1. Обоснование

Геометрия измерений обычно более сложна, чем точка в числовом континууме. Измерение обычно представляет собой отрезок этого континуума с нечёткими границами. Измеряемые показатели выражаются интервалами вследствие неопределённости и случайности, а также того, что измеряемое значение может отражать некоторое условие, например, диапазон температур стабильности протеина.

Руководствуясь только здравым смыслом, кажется более удобным хранить такие данные в виде интервалов, а не в виде двух отдельных чисел. На практике это оказывается даже эффективнее в большинстве приложений.

Более того, вследствие нечёткости границ использование традиционных числовых типов данных приводит к определённой потере информации. Рассмотрим такой пример: ваш инструмент выдаёт 6.50 и вы вводите это значение в базу данных. Что вы получите, прочитав это значение из базы? Смотрите:

```
test=> select 6.50 :: float8 as "pH";
 pH
---
6.5
(1 row)
```

В мире измерений, 6.50 — не то же самое, что 6.5. И разница между этими измерениями иногда бывает критической. Экспериментаторы обычно записывают (и публикуют) цифры, которые заслуживают доверия. Запись 6.50 на самом деле представляет неточный интервал, содержащийся внутри большего и ещё более неточного интервала, 6.5, и единственное, что у них может быть общего, это их центральные точки. Поэтому мы определённо не хотим, чтобы такие разные элементы данных выглядели одинаково.

Вывод? Удобно иметь специальный тип данных, в котором можно сохранить границы интервала с произвольной переменной точностью. В данном случае точность переменная в том смысле, что для каждого элемента данных она может записываться индивидуально.

Проверьте это:

```
test=> select '6.25 .. 6.50'::seg as "pH";
          pH
------------
6.25 .. 6.50
(1 row)
```

F.48.2. Синтаксис

Внешнее представление интервала образуется одним или двумя числами с плавающей точкой, соединёнными оператором диапазона (`..` или `...`). Кроме того, интервал можно задать центральной точкой плюс/минус отклонение. Также этот тип позволяет сохранить

дополнительные индикаторы достоверности (<, > или ~). (Однако индикаторы достоверности игнорируются всеми встроенными операторами.) Допустимые представления показаны в Таблице F.28; некоторые примеры приведены в Таблице F.29.

В Таблице F.28 символы *x*, *y* и *delta* обозначают числа с плавающей точкой. Перед значениями *x* и *y*, но не *delta*, может быть добавлен индикатор достоверности.

Таблица F.28. Внешнее представление seg

x	Одно значение (интервал нулевой длины)
x `..` *y*	Интервал от *x* до *y*
x `(+-)` *delta*	Интервал от *x* - *delta* до *x* + *delta*
x `..`	Открытый интервал с нижней границей *x*
`..` *x*	Открытый интервал с верхней границей *x*

Таблица F.29. Примеры допустимых вводимых значений seg

`5.0`	Создаёт сегмент нулевой длины (или точку, если хотите)
`~5.0`	Создаёт сегмент нулевой длины и записывает ~ в данные. Знак ~ игнорируется при операциях с `seg`, но сохраняется как комментарий.
`<5.0`	Создаёт точку с координатой 5.0. Знак < игнорируется, но сохраняется как комментарий.
`>5.0`	Создаёт точку с координатой 5.0. Знак > игнорируется, но сохраняется как комментарий.
`5(+-)0.3`	Создаёт интервал `4.7 .. 5.3`. Заметьте, что запись `(+-)` не сохраняется.
`50 ..`	Всё, что больше или равно 50
`.. 0`	Всё, что меньше или равно 0
`1.5e-2 .. 2E-2`	Создаёт интервал `0.015 .. 0.02`
`1 ... 2`	То же, что и `1...2`, либо `1 .. 2`, либо `1..2` (пробелы вокруг оператора диапазона игнорируются)

Так как оператор `...` часто используется в источниках данных, он принимается в качестве альтернативного написания оператора `..`. К сожалению, это порождает неоднозначность при разборе: неясно, какая верхняя граница имеется в виду в записи `0...23` — `23` или `0.23`. Для разрешения этой неоднозначности во входных числах `seg` перед десятичной точкой всегда должна быть минимум одна цифра.

В качестве меры предосторожности, `seg` не принимает интервалы с нижней границей, превышающей верхнюю, например: `5 .. 2`.

F.48.3. Точность

Значения `seg` хранятся внутри как пары 32-битных чисел с плавающей точкой. Это значит, что числа с более чем 7 значащими цифрами будут усекаться.

Числа, содержащие 7 и меньше значащих цифр, сохраняют изначальную точность. То есть, если запрос возвращает 0.00, вы можете быть уверены, что конечные нули не являются артефактами форматирования: они отражают точность исходных данных. Количество ведущих нулей не влияет на точность: значение 0.0067 будет считаться имеющим только две значащих цифры.

F.48.4. Использование

Модуль `seg` включает класс операторов индекса GiST для значений `seg`. Операторы, поддерживаемые этим классом операторов, перечислены в Таблице F.30.

Таблица F.30. Операторы seg для GiST

Оператор	Описание
`[a, b] << [c, d]`	[a, b] полностью находится левее [c, d]. То есть, [a, b] << [c, d] — true, если b < c, и false в противном случае.
`[a, b] >> [c, d]`	[a, b] полностью находится правее [c, d]. То есть, [a, b] >> [c, d] — true, если a > d, и false в противном случае.
`[a, b] &< [c, d]`	Пересекает или левее — Ещё лучше это читается как «не простирается правее». Результатом будет true, когда b <= d.
`[a, b] &> [c, d]`	Пересекает или правее — Ещё лучше это читается как «не простирается левее». Результатом будет true, когда a >= c.
`[a, b] = [c, d]`	Равенство — сегменты [a, b] и [c, d] равны, то есть, a = c и b = d.
`[a, b] && [c, d]`	Сегменты [a, b] и [c, d] пересекаются.
`[a, b] @> [c, d]`	Сегмент [a, b] содержит сегмент [c, d], то есть, a <= c и b >= d.
`[a, b] <@ [c, d]`	Сегмент [a, b] содержится в [c, d], то есть, a >= c и b <= d.

(До версии PostgreSQL 8.2 операторы включения `@>` и `<@` обозначались соответственно как `@` и `~`. Эти имена по-прежнему действуют, но считаются устаревшими и в конце концов будут упразднены. Заметьте, что старые имена произошли из соглашения, которому раньше следовали ключевые геометрические типы данных!)

Также поддерживаются стандартные операторы для B-дерева, например:

Оператор	Описание
`[a, b] < [c, d]`	Меньше
`[a, b] > [c, d]`	Больше

Эти операторы не имеют большого смысла ни для какой практической цели, кроме сортировки. Эти операторы сначала сравнивают (a) с (c), и если они равны, сравнивают (b) с (d). Результат сравнения позволяет упорядочить значения образом, подходящим для большинства случаев, что полезно, если вы хотите применять ORDER BY с этим типом.

F.48.5. Замечания

Примеры использования можно увидеть в регрессионном тесте `sql/seg.sql`.

Механизм, преобразующий `(+-)` в обычные диапазоны, не вполне точно определяет число значащих цифр для границ. Например, он добавляет дополнительную цифру к нижней границе, если результирующий интервал включает степень десяти:

```
postgres=> select '10(+-)1'::seg as seg;
      seg
---------
9.0 .. 11             -- должно быть: 9 .. 11
```

Производительность индекса-R-дерева может значительно зависеть от начального порядка вводимых значений. Может быть очень полезно отсортировать входную таблицу по столбцу `seg`; пример можно найти в скрипте `sort-segments.pl`.

F.48.6. Благодарности

Первый автор: Джин Селков мл. <selkovjr@mcs.anl.gov>, Аргоннская национальная лаборатория, Отдел математики и компьютерных наук

Я очень благодарен в первую очередь профессору Джо Геллерштейну (*https://dsf.berkeley.edu/jmh/*) за пояснение сути GiST (*http://gist.cs.berkeley.edu/*). Я также признателен всем разработчикам Postgres в настоящем и прошлом за возможность создать свой собственный мир и спокойно жить в нём. Ещё я хотел бы выразить признательность Аргоннской лаборатории и Министерству энергетики США за годы постоянной поддержки моих исследований в области баз данных.

F.49. sepgsql

Загружаемый модуль `sepgsql` поддерживает мандатное управление доступом (MAC, Mandatory Access Control) с метками, построенное на базе политик безопасности SELinux.

> **Предупреждение**
>
> Текущая реализация имеет существенные ограничения и контролирует не все действия. См. Подраздел F.49.7.

F.49.1. Обзор

Этот модуль интегрируется в SELinux и обеспечивает дополнительный уровень проверок безопасности, расширяющий и дополняющий обычные средства Postgres Pro. С точки зрения SELinux, данный модуль позволяет Postgres Pro выполнять роль менеджера объектов в пространстве пользователя. При выполнении запроса DML обращение к каждой таблице или функции в нём будет контролироваться согласно системной политике безопасности. Эта проверка дополняет штатную проверку разрешений SQL, которую производит Postgres Pro.

Механизм SELinux принимает решения о разрешении доступа на основе меток безопасности, представляемых строками вида `system_u:object_r:sepgsql_table_t:s0`. В каждом решении учитываются две метки: метка субъекта, пытающегося выполнить действие, и метка объекта, над которым должно совершаться это действие. Так как эти метки могут применяться к объекту любого вида, решения о разрешении доступа к объектам внутри базы данных могут подчиняться (и с этим модулем фактически подчиняются) общим критериям, применяемым к объектам любого другого типа, например, к файлам. Эта схема позволяет организовать централизованную политику безопасности для защиты информационных активов, не зависящую от того, как именно хранятся эти активы.

Назначить метку безопасности объекту баз данных позволяет команда SECURITY LABEL.

F.49.2. Установка

Модуль `sepgsql` может работать только в Linux 2.6.28 и новее с включённым SELinux. На остальных платформах он не поддерживается. Вам также понадобится libselinux 2.1.10 или новее и selinux-policy 3.9.13 или новее (хотя в некоторых дистрибутивах необходимые правила могут быть адаптированы к политике старой версии).

Команда `sestatus` позволяет проверить состояние SELinux. Типичный её вывод выглядит так:

```
$ sestatus
SELinux status:                 enabled
SELinuxfs mount:                /selinux
```

```
Current mode:                   enforcing
Mode from config file:          enforcing
Policy version:                 24
Policy from config file:        targeted
```

Если SELinux отключён или не установлен, его необходимо привести в рабочее состояние, прежде чем устанавливать этот модуль.

Чтобы собрать этот модуль, добавьте параметр `--with-selinux` в команду Postgres Pro `configure`. Убедитесь в том, что в момент сборки установлен RPM-пакет `libselinux-devel`.

Чтобы использовать этот модуль, вы должны включить `sepgsql` в shared_preload_libraries в `postgresql.conf`. Этот модуль не будет корректно работать, если загрузить его каким-либо другим способом. Загрузив его, нужно выполнить `sepgsql.sql` в каждой базе данных. Этот скрипт установит функции, необходимые для управления метками безопасности, и назначит начальные метки безопасности.

Следующий пример показывает, как инициализировать новый кластер баз данных и установить в него функции и метки безопасности `sepgsql`. Измените пути в соответствии с размещением вашей инсталляции:

```
$ export PGDATA=/path/to/data/directory
$ initdb
$ vi $PGDATA/postgresql.conf
  изменить
    #shared_preload_libraries = ''                # (после изменения требуется
 перезапуск)
  на
    shared_preload_libraries = 'sepgsql'          # (после изменения требуется
 перезапуск)
$ for DBNAME in template0 template1 postgres; do
    postgres --single -F -c exit_on_error=true $DBNAME \
      </usr/local/pgsql/share/contrib/sepgsql.sql >/dev/null
  done
```

Заметьте, что вы можете увидеть следующие уведомления, в зависимости от конкретных установленных версий libselinux и selinux-policy:

```
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 33 has invalid object type
 db_blobs
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 36 has invalid object type
 db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 37 has invalid object type
 db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 38 has invalid object type
 db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 39 has invalid object type
 db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 40 has invalid object type
 db_language
```

Эти сообщения не критичны и их можно игнорировать.

Если процесс установки завершается без ошибок, вы можете запустить сервер обычным образом.

F.49.3. Регрессионные тесты

Природа SELinux такова, что для проведения регрессионных тестов `sepgsql` требуются дополнительные действия по настройке и некоторые из них должен выполнять root. Регрессионные тесты не будут запускаться обычной командой `make check` или `make installcheck`; вы должны настроить конфигурацию и затем вызвать тестовый скрипт вручную. Тесты должны

запускаться в каталоге `contrib/sepgsql` настроенного дерева сборки Postgres Pro. Хотя им требуется дерево сборки, эти тесты рассчитаны на использование установленного сервера, то есть они примерно соответствуют `make installcheck`, но не `make check`.

Сначала установите `sepgsql` в рабочую базу данных по инструкциям, приведённым в Подразделе F.49.2. Заметьте, что для этого текущий пользователь операционной системы должен подключаться к базе данных как суперпользователь без аутентификации по паролю.

На втором шаге соберите и установите пакет политики для регрессионного теста. Политика `sepgsql-regtest` представляет собой политику особого назначения, предоставляющую набор правил, включаемых во время регрессионных тестов. Её следует скомпилировать из исходного файла `sepgsql-regtest.te`, что можно сделать командой `make` со скриптом Makefile, поставляемым с SELinux. Вам нужно будет найти нужный Makefile в своей системе; путь, показанный ниже, приведён только в качестве примера. Скомпилировав пакет политики, его нужно установить с помощью команды `semodule`, которая загружает переданные ей пакеты в ядро. Если пакет установлен корректно, команда `semodule -l` должна вывести `sepgsql-regtest` в списке доступных пакетов политик:

```
$ cd .../contrib/sepgsql
$ make -f /usr/share/selinux/devel/Makefile
$ sudo semodule -u sepgsql-regtest.pp
$ sudo semodule -l | grep sepgsql
sepgsql-regtest 1.07
```

На третьем шаге включите параметр `sepgsql_regression_test_mode`. По соображениям безопасности, правила в `sepgsql-regtest` по умолчанию неактивны; параметр `sepgsql_regression_test_mode` активирует правила, необходимые для проведения регрессионных тестов. Включить этот параметр можно командой `setsebool`:

```
$ sudo setsebool sepgsql_regression_test_mode on
$ getsebool sepgsql_regression_test_mode
sepgsql_regression_test_mode --> on
```

На четвёртом шаге убедитесь в том, что ваша оболочка работает в домене `unconfined_t`:

```
$ id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

Если необходимо сменить рабочий домен, в подробностях это описывается в Подразделе F.49.8.

Наконец, запустите скрипт регрессионного теста:

```
$ ./test_sepgsql
```

Этот скрипт попытается проверить, все ли шаги по настройке конфигурации выполнены корректно, а затем запустит регрессионные тесты для модуля `sepgsql`.

Завершив тесты, рекомендуется отключить параметр `sepgsql_regression_test_mode`:

```
$ sudo setsebool sepgsql_regression_test_mode off
```

Другой, возможно, более предпочтительный вариант — удалить политику `sepgsql-regtest` полностью:

```
$ sudo semodule -r sepgsql-regtest
```

F.49.4. Параметры GUC

`sepgsql.permissive` (`boolean`)

Этот параметр переводит `sepgsql` в разрешительный режим, вне зависимости от режима системы. По умолчанию он имеет значение off (отключён). Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

Когда этот параметр включён, sepgsql действует в разрешительном режиме, даже если SELinux в целом находится в ограничительном режиме. Этот параметр полезен в первую очередь для тестирования.

sepgsql.debug_audit (boolean)

Этот параметр включает вывод сообщений аудита вне зависимости от параметров системной политики. По умолчанию он отключён (имеет значение off), что означает, что сообщения будут выводиться согласно параметрам системы.

Политики безопасности SELinux также содержит правила, определяющие, будут ли фиксироваться в журнале определённые события. По умолчанию фиксируются нарушения доступа, а успешный доступ — нет.

Этот параметр принудительно включает фиксирование в журнале всех возможных событий, вне зависимости от системной политики.

F.49.5. Функциональные возможности

F.49.5.1. Управляемые классы объектов

Модель безопасности SELinux описывает все правила доступа в виде отношений между сущностью субъекта (обычно, это клиент базы данных) и сущностью объекта (например, объектом базы данных), каждая из которых определяется меткой безопасности. Если осуществляется попытка доступа к непомеченному объекту, он обрабатывается как объект, имеющий метку unlabeled_t.

В настоящее время sepgsql позволяет назначать метки безопасности схемам, таблицам, столбцам, последовательностям, представлениям и функциям. Когда sepgsql активен, метки безопасности автоматически назначаются поддерживаемым объектам базы в момент создания. Такая метка называется меткой безопасности по умолчанию и устанавливается согласно политике безопасности системы, которая учитывает метку создателя, метку, назначенную родительскому объекту создаваемого объекта и, возможно, имя создаваемого объекта.

Новый объект базы, как правило, наследует метку безопасности, назначенную родительскому объекту, если только в политике безопасности не заданы специальные правила, называемые правилами перехода типов (в этом случае может быть назначена другая метка). Для схем родительским объектом является текущая база данных; для таблиц, последовательностей, представлений и функций — схема, содержащая эти объекты; для столбцов — таблица.

F.49.5.2. Разрешения для DML

Для таблиц, задействованных в запросе в качестве целевых, проверяются разрешения db_table:select, db_table:insert, db_table:update или db_table:delete в зависимости от типа оператора; кроме того, для всех таблиц, содержащих столбцы, фигурирующие в предложении WHERE или RETURNING, или служащих источником данных для UPDATE и т. п., также проверяется разрешение db_table:select.

Для всех задействованных столбцов также проверяются разрешения на уровне столбцов. Разрешение db_column:select проверяется не только для столбцов, которые считываются оператором SELECT, но и для тех, к которым обращаются другие операторы DML; db_column:update или db_column:insert также проверяется для столбцов, изменяемых операторами UPDATE или INSERT.

Например, рассмотрим запрос:

```
UPDATE t1 SET x = 2, y = md5sum(y) WHERE z = 100;
```

В данном случае db_column:update будет проверяться для столбца t1.x, так как он изменяется, db_column:{select update} будет проверяться для t1.y, так как он и считывается, и изменяется, а db_column:select — для столбца t1.z, так как он только считывается. На уровне таблицы также будет проверяться разрешение db_table:{select update}.

Для последовательностей проверяется разрешение db_sequence:get_value, когда имеет место обращение к объекту последовательности в SELECT; заметьте, однако, что в настоящее время разрешения на выполнение связанных функций, таких как, lastval(), не проверяются.

Для представлений проверяется db_view:expand, а затем все другие соответствующие разрешения для объектов, развёрнутых из определения представления, в индивидуальном порядке.

Для функций проверяется db_procedure:{execute}, когда пользователь пытается выполнить функцию в составе запроса, либо при вызове по быстрому пути. Если эта функция является доверенной процедурой, также проверяется разрешение db_procedure:{entrypoint}, чтобы удостовериться, что эта функция может быть точкой входа в доверенную процедуру.

При обращении к любому объекту схемы необходимо иметь разрешение db_schema:search для содержащей его схемы. Когда имя целевого объекта не дополняется схемой, схемы, для которых данное разрешение отсутствует, не будут просматриваться (то же происходит, если у пользователя нет права USAGE для этой схемы). Когда схема указывается явно, пользователь получит ошибку, если он не имеет требуемого разрешения для доступа к указанной схеме.

Клиенту должен быть разрешён доступ ко всем задействованным в запросе таблицам и столбцам, даже если они проявились в нём в результате разворачивания представлений, так что правила применяются согласованно вне зависимости от варианта обращения к содержимому таблиц.

Стандартная система привилегий позволяет суперпользователям баз данных изменять системные каталоги с помощью команд DML и обращаться к таблицам TOAST или модифицировать их. Когда модуль sepgsql активен, эти операции запрещаются.

F.49.5.3. Разрешения для DDL

SELinux определяет набор разрешений для управления стандартными операциями для каждого типа объекта: создание, изменение определения, удаление и смена метки безопасности. В дополнение к ним для некоторых типов объектов предусмотрены специальные разрешения для управления их специфическими операциями, как например, добавление или удаление объектов в определённой схеме.

Для создания нового объекта базы данных требуется разрешение create. SELinux разрешает или запрещает выполнение этой операции в зависимости от метки безопасности клиента и предполагаемой метки безопасности нового объекта. В некоторых случаях требуются дополнительные разрешения:

- CREATE DATABASE дополнительно требует разрешения getattr в исходной или шаблонной базе данных.
- Создание объекта схемы дополнительно требует разрешения add_name в родительской схеме.
- Создание таблицы дополнительно требует разрешения на создание каждой отдельного столбца таблицы, как если бы каждый столбец таблицы был отдельным объектом верхнего уровня.
- Создание функции с атрибутом LEAKPROOF дополнительно требует разрешения install. (Это разрешение также проверяется, когда атрибут LEAKPROOF устанавливается для существующей функции.)

Когда выполняется команда DROP, для удаляемого объекта будет проверяться разрешение drop. Разрешения будут также проверяться и для объектов, удаляемых косвенно, вследствие указания CASCADE. Для удаления объектов, содержащихся в определённой схеме, (таблиц, представления, последовательностей и процедур) дополнительно нужно иметь разрешение remove_name в этой схеме.

Когда выполняется команда ALTER, для каждого модифицируемого объекта проверяется разрешение setattr, кроме подчинённых объектов, таких как индексы или триггеры таблиц (на них распространяются разрешения родительского объекта). В некоторых случаях требуются дополнительные разрешения:

- При перемещении объекта в новую схему дополнительно требуется разрешение `remove_name` в старой схеме и `add_name` в новой.
- Для установки атрибута `LEAKPROOF` для функции требуется разрешение `install`.
- Для использования SECURITY LABEL дополнительно требуется разрешение `relabelfrom` для объекта с его старой меткой безопасности и `relabelto` для этого объекта с новой меткой безопасности. (В случаях, когда установлено несколько поставщиков меток и пользователь пытается задать метку, неподконтрольную SELinux, должно проверяться только разрешение `setattr`. В настоящее время этого не происходит из-за ограничений реализации.)

F.49.5.4. Доверенные процедуры

Доверенные процедуры похожи на функции, определяющие контекст безопасности, или команды setuid. В SELinux реализована возможность запускать доверенный код с меткой безопасности, отличной от метки клиента, как правило, для предоставления чётко контролируемого доступа к важным данным (при этом например, могут отсеиваться строки или хранимые значения могут выводиться с меньшей точностью). Будет ли функция вызываться как доверенная процедура, определяется её меткой безопасности и политикой операционной системы. Например:

```
postgres=# CREATE TABLE customer (
               cid     int primary key,
               cname   text,
               credit  text
           );
CREATE TABLE
postgres=# SECURITY LABEL ON COLUMN customer.credit
               IS 'system_u:object_r:sepgsql_secret_table_t:s0';
SECURITY LABEL
postgres=# CREATE FUNCTION show_credit(int) RETURNS text
             AS 'SELECT regexp_replace(credit, ''-[0-9]+$'', ''-xxxx'', ''g'')
                         FROM customer WHERE cid = $1'
           LANGUAGE sql;
CREATE FUNCTION
postgres=# SECURITY LABEL ON FUNCTION show_credit(int)
               IS 'system_u:object_r:sepgsql_trusted_proc_exec_t:s0';
SECURITY LABEL
```

Показанные выше операции должен выполнять пользователь с правами администратора.

```
postgres=# SELECT * FROM customer;
ERROR:  SELinux: security policy violation
postgres=# SELECT cid, cname, show_credit(cid) FROM customer;
 cid | cname  |     show_credit
-----+--------+---------------------
   1 | taro   | 1111-2222-3333-xxxx
   2 | hanako | 5555-6666-7777-xxxx
(2 rows)
```

В данном случае обычный пользователь не может обращаться к `customer.credit` напрямую, но доверенная процедура `show_credit` позволяет ему получить номера кредитных карт клиентов, в которых будут скрыты некоторые цифры.

F.49.5.5. Динамические переключения домена

Возможность динамического перехода из домена в домен SELinux позволяет переводить метку безопасности клиентского процесса, клиентский домен в новый контекст, если это допускается политикой безопасности. Для этого клиент должен иметь разрешение `setcurrent`, а также разрешение `dyntransition` для перехода из старого в новый домен.

Динамические переключения домена следует тщательно продумывать, так как таким образом пользователи могут менять свои метки, а значит и привилегии, по собственному желанию, а не

(как в случае с доверенными процедурами) по правилам, диктуемым системой. Таким образом, разрешение `dyntransition` считается безопасным, только когда применяется для переключения в домен с более ограниченным набором привилегий, чем текущий. Например:

```
regression=# select sepgsql_getcon();
                    sepgsql_getcon
-----------------------------------------------------
 unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
(1 row)

regression=# SELECT sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-
s0:c1.c4');
 sepgsql_setcon
----------------
 t
(1 row)

regression=# SELECT sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-
s0:c1.c1023');
ERROR:  SELinux: security policy violation
```

В показанном выше примере мы смогли переключиться из более широкого диапазона MCS `c1.c1023` в более узкий `c1.c4`, но переключение в обратную сторону было запрещено.

Сочетание динамического переключения домена с доверенными процедурами позволяет получить интересное решение, подходящее для реализации жизненного цикла процессов с пулом соединений. Даже если вашему менеджеру пула соединений не разрешается запускать многие команды SQL, вы можете разрешить ему сменить метку безопасности клиента, вызвав функцию `sepgsql_setcon()` из доверенной процедуры; для этого может передаваться удостоверение для авторизации запроса на смену метки клиента. После этого сеанс получит привилегии целевого пользователя, а не пользователя пула соединений. Позднее менеджер пула может отменить смену контекста безопасности, вызвав `sepgsql_setcon()` с аргументом `NULL`, так же из доверенной процедуры с необходимыми проверками разрешений. Идея этого подхода в том, что только этой доверенной процедуре будет разрешено менять действующую метку безопасности и только в том случае, когда ей передаётся правильное удостоверение. Разумеется, чтобы это решение было безопасным, хранилище удостоверений (таблица, определение процедуры или что-то другое) не должно быть общедоступным.

F.49.5.6. Разное

Выполнение команды LOAD в активном режиме запрещается, так как любой загруженный модуль может легко обойти ограничения политики безопасности.

F.49.6. Функции sepgsql

В Таблице F.31 перечислены все доступные функции.

Таблица F.31. Функции sepgsql

`sepgsql_getcon() returns text`	Возвращает клиентский домен, текущую метку безопасности клиента.
`sepgsql_setcon(text) returns bool`	Переключает домен клиента текущего сеанса в новый домен, если это допускает политика безопасности. Эта функция также принимает в аргументе `NULL` как запрос на переход в начальный домен клиента.
`sepgsql_mcstrans_in(text) returns text`	Переводит заданный диапазон MLS/MCS из полной записи в низкоуровневый формат, если работает демон mcstrans.

`sepgsql_mcstrans_out(text) returns text`	Переводит заданный диапазон MLS/MCS из низкоуровневого формата в полную запись, если работает демон mcstrans.
`sepgsql_restorecon(text) returns bool`	Устанавливает начальные метки безопасности для всех объектов в текущей базе данных. В аргументе может передаваться NULL или имя файла со спецификациями контекстов, который будет применяться вместо стандартного системного.

F.49.7. Ограничения

Разрешения для языка определения данных (DDL, Data Definition Language)

Вследствие ограничений реализации, для некоторых операций DDL разрешения не проверяются.

Разрешения для языка управления данными (DCL, Data Control Language)

Вследствие ограничений реализации, для операций DCL разрешения не проверяются.

Управление доступом на уровне строк

Postgres Pro поддерживает ограничение доступа на уровне строк, а `sepgsql` — нет.

Скрытые каналы

Модуль `sepgsql` не пытается скрыть существование определённого объекта, даже если пользователю не разрешено обращаться к нему. Например, возможно догадаться о существовании невидимого объекта по конфликтам первичного ключа, нарушениям внешних ключей и т. д., даже когда нельзя получить содержимое этого объекта. Существование совершенно секретной таблицы невозможно скрыть; надеяться можно только на то, что будет защищено её содержимое.

F.49.8. Внешние ресурсы

SE-PostgreSQL Introduction, Введение в SE-PostgreSQL

На этой вики-странице даётся краткий обзор этого решения и рассказывается об архитектуре и конструкции безопасности, администрировании и ожидаемых в будущем возможностях.

SELinux User's and Administrator's Guide, Руководство пользователя и администратора SELinux

В этом документе представлен широкий спектр знаний по администрированию SELinux в ОС. В первую очередь он ориентирован на системы Red Hat, но его область применения не ограничена ими.

Fedora SELinux FAQ, Часто задаваемые вопросы по SELinux в ОС Fedora

В этом документе даются ответы на часто задаваемые вопросы по SELinux. В первую очередь он ориентирован на ОС Fedora, но его область применения не ограничена ей.

F.49.9. Автор

КайГай Кохэй `<kaigai@ak.jp.nec.com>`

F.50. shared_ispell

Модуль `shared_ispell` предоставляет разделяемый словарь ispell, то есть словарь, расположенный в общем сегменте памяти. В традиционной реализации ispell каждый сеанс инициализирует и загружает собственный экземпляр словаря, и на это уходит много ресурсов процессора и памяти.

Это расширение выделяет область памяти в разделяемом сегменте (вы должны выбрать его размер заранее) и загружает в неё словарь при первом использовании.

F.50.1. Функции

Реализованные в модуле `shared_ispell` функции перечислены в Таблице F.32.

Таблица F.32. Функции `shared_ispell`

Функция	Возвращает	Описание
`shared_ispell_reset()`	`void`	Сбрасывает словари (например, для повторной загрузки изменённых файлов с диска). Сеансы, в которых словари уже использовались, будут должны переинициализировать их.
`shared_ispell_mem_used()`	`int`	Возвращает объём памяти, используемой загруженными словарями в разделяемом сегменте (в байтах).
`shared_ispell_mem_available()`	`int`	Возвращает объём доступной памяти в разделяемом сегменте.
`shared_ispell_dicts()`	`setof(dict_name varchar, affix_name varchar, words int, affixes int, bytes int)`	Возвращает список словарей, загруженных в разделяемый сегмент.
`shared_ispell_stoplists()`	`setof(stop_name varchar, words int, bytes int)`	Возвращает список наборов стоп-слов, загруженных в разделяемый сегмент.

F.50.2. Параметры GUC

`shared_ispell.max_size` (`int`)

Определяет максимальный размер разделяемого сегмента. Это жёсткий предел; разделяемый сегмент нельзя расширить, поэтому нужно установить такой размер, чтобы в этом сегменте уместились все словари и при этом он не занимал лишний объём.

F.50.3. Использование словаря

Этот модуль должен выделить область памяти в разделяемом сегменте. Поэтому добавьте в файл конфигурации следующие параметры (или отредактируйте текущие значения):

```
# загружаемые библиотеки
shared_preload_libraries = 'shared_ispell'

# объём разделяемой памяти
shared_ispell.max_size = 32MB
```

Чтобы понять, сколько памяти вам действительно нужно, задайте сначала большое значение (например, 200 МБ) и загрузите все нужные вам словари. Затем воспользуйтесь функцией `shared_ispell_mem_used()` и определите, сколько памяти на самом деле используется (и скорректируйте соответствующую переменную GUC `shared_ispell.max_size`).

Не задавайте в точности полученное значение, оставьте небольшой резерв, чтобы вы могли перезагружать словари, не изменяя предел GUC max_size (что потребует перезапуска БД). Объёма в районе 512 КБ будет вполне достаточно.

Это расширение определяет шаблон `shared_ispell`, используя который можно определять собственные словари текстового поиска. Например, вы можете сделать следующее:

```
CREATE TEXT SEARCH DICTIONARY english_shared (
```

```
    TEMPLATE = shared_ispell,
    DictFile = en_us,
    AffFile = en_us,
    StopWords = english
);

CREATE TEXT SEARCH CONFIGURATION public.english_shared
    ( COPY = pg_catalog.simple );

ALTER TEXT SEARCH CONFIGURATION english_shared
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                      word, hword, hword_part
    WITH english_shared, english_stem;
```

Проверить полученную конфигурацию можно так:

```
SELECT * FROM ts_debug('english_shared', 'abilities');
   alias   |   description   |   token   |         dictionaries          |   dictionary
   |  lexemes
-----------+-----------------+-----------+-------------------------------
+----------------+-----------
 asciiword | Word, all ASCII | abilities | {english_shared,english_stem} |
 english_shared | {ability}
(1 row)
```

Вы также можете изменить существующую конфигурацию текстового поиска. Например, если у вас есть конфигурация `public.english`, вы можете подключить к ней шаблон `shared_ispell`:

```
ALTER TEXT SEARCH CONFIGURATION public.english
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                      word, hword, hword_part
    WITH english_shared, english_stem;
```

F.50.4. Автор

Томаш Вондра `<tomas.vondra@2ndquadrant.com>`, Прага, Чешская республика

F.51. snowball_ext

Модуль `snowball_ext` реализует поддержку полнотекстового поиска для непальского языка в словаре Snowball. Этот модуль добавляет стеммер `nepali_stem`, конфигурацию полнотекстового поиска `nepali`, а также создаёт шаблон `snowball_ext`.

F.51.1. Установка

Модуль `snowball_ext` включён в состав Postgres Pro. Установив Postgres Pro, создайте расширение `snowball_ext`, выполнив следующую команду:

```
CREATE EXTENSION snowball_ext;
```

F.51.2. Использование

Функциональность расширения `snowball_ext` для непальского языка аналогично функциональности встроенного словаря Snowball для других поддерживаемых языков. Подробнее о настройке полнотекстового поиска можно узнать в Главе 12.

F.51.3. Авторы

Postgres Professional, Москва, Россия

F.52. sr_plan

Важно

Поддержка расширения sr_plan в Postgres Pro прекращена.

F.52.1. Обоснование

Расширение sr_plan позволяет сохранять планы выполнения запросов и использовать эти планы при последующем выполнении тех же запросов, исключая многократную оптимизацию одного и того же запроса.

Модуль sr_plan действует подобно системе Oracle Outline. Используя его, вы можете жёстко зафиксировать план выполнения. Это бывает необходимо, если вы уверены, что планировщик не сможет выработать лучший план.

Обычно системные администраторы играют с запросами интерактивно и сохраняют выбранные планы для использования с запросами, время обработки которых должно быть предсказуемым.

Затем сохранённые планы должны задействоваться приложением, выполняющим эти запросы.

F.52.2. Установка

Выполните в вашей базе:

```
CREATE EXTENSION sr_plan;
```

И отредактируйте postgresql.conf:

```
shared_preload_libraries = 'sr_plan'
```

Важно, чтобы эта библиотека предзагружалась при запуске сервера, так как применение сохранённых планов включается на уровне баз данных и не зависит от каких-либо параметров соединений.

F.52.3. Использование

Если вы хотите сохранить план запроса, необходимо установить переменную:

```
set sr_plan.write_mode = true;
```

После этого планы всех последующих запросов будут сохраняться в таблице sr_plans, пока этой переменной не будет присвоено значение false. Помните, что при этом сохраняются планы всех запросов, включая повторные. Выполните нужный вам запрос:

```
select query_hash from sr_plans where query_hash=10;
```

Отключите сохранение планов:

```
set sr_plan.write_mode = false;
```

Убедитесь в том, что план запроса сохранён:

```
select query_hash, enable, valid, query, explain_jsonb_plan(plan) from sr_plans;

 query_hash | enable | valid |                          query                          |
               explain_jsonb_plan
------------+--------+-------+---------------------------------------------------------
+-------------------------------------------------
 1783086253 | f      | t     | select query_hash from sr_plans where query_hash=10; |
 Bitmap Heap Scan on sr_plans                        +
            |        |       |                                                        |
 Recheck Cond: (query_hash = 10)                  +
            |        |       |                                                        |
 ->  Bitmap Index Scan on sr_plans_query_hash_idx+
```

```
      |        |       |                                                  |
Index Cond: (query_hash = 10)             +
      |        |       |                                                  |
```

Обратите внимание на вызов функции `explain_jsonb_plan`, которая позволяет визуализировать план выполнения подобно тому, как это делает команда EXPLAIN.

Планы сохраняются в базе данных в виде jsonb. По умолчанию все записанные планы отключены, их использование необходимо разрешить вручную:

Чтобы разрешить использование сохранённого плана, выполните:

```
update sr_plans set enable=true where query_hash=1783086253;
```

(Здесь 1783086253 указано только для примера.) После этого при выполнении данного запроса будет использоваться план из таблицы `sr_plans`.

Кроме того, sr_plan позволяет сохранять планы параметризованных запросов. В этом случае в запросе будут некоторые константы, которые, как предполагается, не должны влиять на план.

Мы можем пометить в режиме сохранения плана эти константы как параметры запроса, используя специальную функцию `_p (anyelement)`. Например:

```
=>create table test_table (a numeric, b text);
CREATE TABLE
=>insert into test_table values (1,'1'),(2,'2'),(3,'3');
INSERT 0 3
=> set sr_plan.write_mode = true;
SET
=> select a,b  from test_table where a = _p(1);
 a | b
---+---
 1 | 1
(1 row)

=> set sr_plan.write_mode = false;
SET
```

Теперь план запроса к нашей таблице сохранён с параметром. И если мы включим сохранённый план, он будет использоваться для запросов с любым значением a, если это значение будет завёрнуто в вызов функции `_p()`.

```
=>update sr_plans set enable = true where query=
  'select a,b from test_table where a = _p(1)';
UPDATE 1
-- Эти запросы будут использовать сохранённый план

=>select a,b from test_table where a = _p(2);
 a | b
---+---
 2 | 2
(1 row)

=>select a,b from test_table where a = _p(3);
 a | b
---+---
 3 | 3
(1 row)

-- Этот запрос не будет использовать сохранённый план, так как
-- константа не завёрнута в _p()
```

```
=>select a,b from test_table where a = 1;
 a | b
---+---
 1 | 1
(1 row)
```

F.53. sslinfo

Модуль `sslinfo` выдаёт информацию о SSL-сертификате, который был представлен текущим клиентом при подключении к Postgres Pro. Этот модуль бесполезен (большинство функций возвратят NULL), если для текущего подключения не задействуется SSL.

Это расширение не будет собираться, если конфигурация была произведена без ключа `--with-openssl`.

F.53.1. Предоставляемые функции

`ssl_is_used() returns boolean`

Возвращает TRUE, если текущее подключение использует SSL, и FALSE в противном случае.

`ssl_version() returns text`

Возвращает имя протокола, по которому организовано SSL-подключение (например TLSv1.0, TLSv1.1 или TLSv1.2).

`ssl_cipher() returns text`

Возвращает имя шифра, используемого для SSL-подключения (например, DHE-RSA-AES256-SHA).

`ssl_client_cert_present() returns boolean`

Возвращает TRUE, если текущий клиент предоставил серверу действительный клиентский SSL-сертификат, и FALSE в противном случае. (Сервер может требовать, а может и не требовать предоставления клиентского сертификата.)

`ssl_client_serial() returns numeric`

Возвращает серийный номер текущего клиентского сертификата. Сочетание серийного номера сертификата с выдавшим его центром сертификации гарантирует однозначную идентификацию сертификата (но не его владелец — владелец должен регулярно менять свои ключи и получать сертификаты в центре сертификации).

Поэтому, если вы используете собственный ЦС и настроили сервер, чтобы он принимал сертификаты только от этого ЦС, серийный номер будет наиболее надёжным (хотя не очень запоминающимся) ключом идентификации пользователя.

`ssl_client_dn() returns text`

Возвращает полное имя субъекта из текущего клиентского сертификата, преобразуя символьные данные в кодировку текущей базы данных. Предполагается, что если в именах в сертификатах используются символы вне таблицы ASCII, то ваша база данных может представить эти символы. Если в вашей базе используется кодировка SQL_ASCII, символы вне ASCII в имени будут представлены последовательностями UTF-8.

Результат выглядит примерно так: `/CN=Somebody /C=Some country/O=Some organization`.

`ssl_issuer_dn() returns text`

Возвращает полное имя издателя текущего клиентского сертификата, преобразуя символьные данные в кодировку текущей базы данных. Преобразования кодировки осуществляются так же, как и в `ssl_client_dn`.

Сочетание возвращаемого значения этой функции с серийным номером сертификата однозначно идентифицирует сертификат.

Эта функция полезна, только если в файле `root.crt` вашего сервера содержатся сертификаты нескольких ЦС, или если один ЦС выдаёт сертификаты для промежуточных центров сертификации.

`ssl_client_dn_field(fieldname text) returns text`

Эта функция возвращает значение указанного поля данных субъекта сертификата, либо NULL, если это поле отсутствует. Имена полей задаются строковыми константами, которые затем преобразуются в идентификаторы объектов ASN1, используя базу данных объектов OpenSSL. Принимаются следующие значения:

```
commonName (или CN)
surname (или SN)
name
givenName (или GN)
countryName (или C)
localityName (или L)
stateOrProvinceName (или ST)
organizationName (или O)
organizationalUnitName (или OU)
title
description
initials
postalCode
streetAddress
generationQualifier
description
dnQualifier
x500UniqueIdentifier
pseudonym
role
emailAddress
```

Все эти поля являются необязательными, за исключением `commonName`. Какие из них будут включены в сертификат, а какие нет, зависит полностью от политики вашего ЦС. Значение этих полей, однако, строго определено стандартами X.500 и X.509, так что их нельзя интерпретировать произвольным образом.

`ssl_issuer_field(fieldname text) returns text`

То же, что `ssl_client_dn_field`, но для издателя, а не для субъекта сертификата.

`ssl_extension_info() returns setof record`

Предоставляет информацию о расширениях клиентского сертификата: имя расширения, значение расширения и является ли это расширение критическим.

F.53.2. Автор

Виктор Вагнер `<vitus@cryptocom.ru>`, ООО «Криптоком»

Дмитрий Воронин `<carriingfate92@yandex.ru>`

Электронный адрес группы разработчиков OpenSSL в Криптокоме: `<openssl@cryptocom.ru>`

F.54. tablefunc

Модуль `tablefunc` содержит ряд функций, возвращающих таблицы (то есть, множества строк). Эти функции полезны и сами по себе, и как примеры написания на C функций, возвращающих наборы строк.

F.54.1. Предоставляемые функции

Функции, предоставляемые модулем `tablefunc`, перечислены в Таблице F.33.

Таблица F.33. Функции `tablefunc`

Функция	Возвращает	Описание
`normal_rand(int numvals, float8 mean, float8 stddev)`	`setof float8`	Выдаёт набор случайных значений, имеющих нормальное распределение
`crosstab(text sql)`	`setof record`	Выдаёт «повёрнутую таблицу», содержащую имена строк плюс *N* столбцов значений, где *N* определяется видом строк, заданным в вызывающем запросе
`crosstabN(text sql)`	`setof table_crosstab_ N`	Выдаёт «повёрнутую таблицу», содержащую имена строк плюс *N* столбцов значений. Функции `crosstab2`, `crosstab3` и `crosstab4` предопределены, но вы можете создать дополнительные функции `crosstabN`, как описано ниже
`crosstab(text source_sql, text category_sql)`	`setof record`	Выдаёт «повёрнутую таблицу» со столбцами значений, заданными вторым запросом
`crosstab(text sql, int N)`	`setof record`	Устаревшая версия `crosstab(text)`. Параметр *N* теперь игнорируется, так как число столбцов значений всегда определяется вызывающим запросом
`connectby(text relname, text keyid_fld, text parent_keyid_fld [, text orderby_fld ], text start_with, int max_depth [, text branch_delim ])`	`setof record`	Выдаёт представление иерархической древовидной структуры

F.54.1.1. `normal_rand`

```
normal_rand(int numvals, float8 mean, float8 stddev) returns setof float8
```

Функция `normal_rand` выдаёт набор случайных значений, имеющих нормальное распределение (распределение Гаусса).

Параметр *numvals* задаёт количество значений, которое выдаст эта функция. Параметр *mean* задаёт медиану нормального распределения, а *stddev* — стандартное отклонение.

Например, этот вызов запрашивает 1000 значений с медианой 5 и стандартным отклонением 3:

```
test=# SELECT * FROM normal_rand(1000, 5, 3);
     normal_rand
---------------------
     1.56556322244898
     9.10040991424657
```

```
     5.36957140345079
   -0.369151492880995
    0.283600703686639
       .
       .
       .
    4.82992125404908
    9.71308014517282
    2.49639286969028
(1000 rows)
```

F.54.1.2. crosstab(text)

```
crosstab(text sql)
crosstab(text sql, int N)
```

Функция `crosstab` применяется для формирования «повёрнутых» отображений, в которых данные идут вдоль строк, а не сверху вниз. Например, мы можем иметь такие данные:

```
row1    val11
row1    val12
row1    val13
...
row2    val21
row2    val22
row2    val23
...
```

и хотим видеть их так:

```
row1    val11   val12   val13   ...
row2    val21   val22   val23   ...
...
```

Функция `crosstab` принимает в текстовом параметре SQL-запрос, выдающий исходные данные первым способом, и выдаёт таблицу, отформатированную вторым способом.

В параметре *`sql`* передаётся SQL-запрос, выдающий исходный набор данных. Этот запрос должен возвращать один столбец `row_name`, один столбец `category` и один столбец `value`. Параметр *`N`* является устаревшим и игнорируется, если передаётся при вызове (раньше он должен был соответствовать количеству выходных столбцов значений, но теперь это количество определяется вызывающим запросом).

Например, заданный запрос может выдавать такой результат:

```
 row_name    cat    value
----------+-------+-------
  row1      cat1    val1
  row1      cat2    val2
  row1      cat3    val3
  row1      cat4    val4
  row2      cat1    val5
  row2      cat2    val6
  row2      cat3    val7
  row2      cat4    val8
```

Функция `crosstab` объявлена как возвращающая `setof record`, так что фактические имена и типы столбцов должны определяться в предложении `FROM` вызывающего оператора `SELECT`, например так:

```
SELECT * FROM crosstab('...') AS ct(row_name text, category_1 text, category_2 text);
```

Этот запрос выдаст примерно такой результат:

```
          <== столбцы значений ==>
 row_name   category_1   category_2
----------+------------+------------
  row1        val1         val2
  row2        val5         val6
```

Предложение `FROM` должно определять результат со столбцом `row_name` (того же типа данных, что у первого результирующего столбца SQL-запроса), за которым следуют N столбцов значений (все того же типа данных, что и третий результирующий столбец SQL-запроса). Количество выходных столбцов значений может быть произвольным и имена выходных столбцов определяете вы сами.

Функция `crosstab` выдаёт одну выходную строку для каждой последовательной группы с одним значением `row_name`. Она заполняет столбцы значений слева направо полями `value` из этих строк. Если в группе оказывается меньше строк, чем выходных столбцов значений, дополнительные столбцы принимают значения NULL; если же строк оказывается больше, лишние строки игнорируются.

На практике в SQL-запросе всегда должно указываться `ORDER BY 1,2`, чтобы входные строки были отсортированы должным образом, то есть, чтобы данные с одинаковым значением `row_name` собирались вместе и корректно упорядочивались в строке. Заметьте, что сама `crosstab` не учитывает второй столбец результата запроса; он присутствует только для того, чтобы определять порядок, в котором значения третьего столбца будут следовать в строке.

Полный пример:

```
CREATE TABLE ct(id SERIAL, rowid TEXT, attribute TEXT, value TEXT);
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att1','val1');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att2','val2');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att3','val3');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att4','val4');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att1','val5');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att2','val6');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att3','val7');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att4','val8');

SELECT *
FROM crosstab(
  'select rowid, attribute, value
   from ct
   where attribute = ''att2'' or attribute = ''att3''
   order by 1,2')
AS ct(row_name text, category_1 text, category_2 text, category_3 text);

 row_name | category_1 | category_2 | category_3
----------+------------+------------+------------
 test1    | val2       | val3       |
 test2    | val6       | val7       |
(2 rows)
```

Вы можете в любом случае обойтись без написания предложения `FROM`, определяющего выходные столбцы, создав собственную функцию crosstab, в определении которой будет зашит желательный тип выходной строки. Это описывается в следующем разделе. Также имеется возможность включить требуемое предложение `FROM` в определение представления.

> **Примечание**
>
> Также изучите команду `\crosstabview` в psql, реализующую функциональность, подобную `crosstab()`.

F.54.1.3. crosstab*N*(text)

```
crosstabN(text sql)
```

Функции crosstab*N* являются примерами того, как можно создать собственные обёртки универсальной функции crosstab, чтобы не приходилось выписывать имена и типы столбцов в вызывающем запросе SELECT. Модуль tablefunc включает функции crosstab2, crosstab3 и crosstab4, определяющие типы выходных строк так:

```
CREATE TYPE tablefunc_crosstab_N AS (
    row_name TEXT,
    category_1 TEXT,
    category_2 TEXT,
        .
        .
        .
    category_N TEXT
);
```

Таким образом, эти функции могут применяться непосредственно, когда входной запрос выдаёт столбцы row_name и value типа text и вы хотите получить на выходе 2, 3 или 4 столбца значений. В остальном эти функции ведут себя в точности так же, как и универсальная функция crosstab.

Так, пример, приведённый в предыдущем разделе, можно переписать и в таком виде:

```
SELECT *
FROM crosstab3(
  'select rowid, attribute, value
   from ct
   where attribute = ''att2'' or attribute = ''att3''
   order by 1,2');
```

Эти функции представлены в основном в демонстрационных целях. Вы можете создать собственные типы возвращаемых данных и реализовать функции на базе нижележащей функции crosstab(). Это можно сделать двумя способами:

- Создать составной тип, описывающий желаемые выходные столбцы, примерно как это делается в примерах в contrib/tablefunc/tablefunc--1.0.sql. Затем нужно выбрать уникальное имя для функции, принимающей один параметр text и возвращающей setof имя_вашего_типа, и связать его с той же нижележащей функцией crosstab на C. Например, если ваш источник данных выдаёт имена строк типа text и значения типа float8, и вы хотите получить 5 столбцов значений:

  ```
  CREATE TYPE my_crosstab_float8_5_cols AS (
      my_row_name text,
      my_category_1 float8,
      my_category_2 float8,
      my_category_3 float8,
      my_category_4 float8,
      my_category_5 float8
  );

  CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(text)
      RETURNS setof my_crosstab_float8_5_cols
      AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE STRICT;
  ```

- Использовать выходные параметры (OUT), чтобы явно определить возвращаемый тип. Тот же пример можно реализовать и таким способом:

  ```
  CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(
      IN text,
      OUT my_row_name text,
  ```

```
    OUT my_category_1 float8,
    OUT my_category_2 float8,
    OUT my_category_3 float8,
    OUT my_category_4 float8,
    OUT my_category_5 float8)
  RETURNS setof record
  AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE STRICT;
```

F.54.1.4. crosstab(text, text)

```
crosstab(text source_sql, text category_sql)
```

Основное ограничение формы `crosstab` с одним параметром состоит в том, что она воспринимает все значения в группе одинаково и вставляет очередное значение в первый свободный столбец. Если вы хотите, чтобы столбцы значений соответствовали определённым категориям данных и некоторые группы могли содержать данные не для всех категорий, этот подход не будет работать. Форма `crosstab` с двумя параметрами решает эту задачу, принимая явный список категорий, соответствующих выходным столбцам.

В параметре *`source_sql`* передаётся SQL-оператор, выдающий исходный набор данных. Этот оператор должен выдавать строки со столбцом `row_name`, столбцом `category` и столбцом `value`. Также он может выдать один или несколько «дополнительных» столбцов. Столбец `row_name` должен быть первым, а столбцы `category` и `value` — последними двумя, именно в этом порядке. Все столбцы между `row_name` и `category` воспринимаются как «дополнительные». Ожидается, что «дополнительные» столбцы будут содержать одинаковые значения для всех строк с одним значением `row_name`.

Например, *`source_sql`* может выдать такой набор данных:

```
SELECT row_name, extra_col, cat, value FROM foo ORDER BY 1;

 row_name    extra_col   cat    value
----------+------------+-----+---------
  row1         extra1    cat1    val1
  row1         extra1    cat2    val2
  row1         extra1    cat4    val4
  row2         extra2    cat1    val5
  row2         extra2    cat2    val6
  row2         extra2    cat3    val7
  row2         extra2    cat4    val8
```

В параметре *`category_sql`* передаётся оператор SQL, выдающий набор категорий. Этот оператор должен возвращать всего один столбец. Он должен выдать минимум одну строку; в противном случае произойдёт ошибка. Кроме того, выдаваемые им значения не должны повторяться, иначе так же произойдёт ошибка. В качестве *`category_sql`* можно передать, например, такой запрос:

```
SELECT DISTINCT cat FROM foo ORDER BY 1;
    cat
  -------
    cat1
    cat2
    cat3
    cat4
```

Функция `crosstab` объявлена как возвращающая тип `setof record`, так что фактические имена и типы выходных столбцов должны определяться в предложении `FROM` вызывающего оператора `SELECT`, например так:

```
SELECT * FROM crosstab('...', '...')
    AS ct(row_name text, extra text, cat1 text, cat2 text, cat3 text, cat4 text);
```

При этом будет получен примерно такой результат:

```
                <==  столбцы значений   ==>
row_name   extra   cat1   cat2   cat3   cat4
---------+-------+------+------+------+------
  row1     extra1  val1   val2          val4
  row2     extra2  val5   val6   val7   val8
```

В предложении `FROM` должно определяться нужное количество выходных столбцов соответствующих типов данных. Если запрос *`source_sql`* выдаёт *`N`* столбцов, первые *`N`*-2 из них должны соответствовать первым *`N`*-2 выходным столбцам. Оставшиеся выходные столбцы должны иметь тип последнего столбца результата *`source_sql`* и их должно быть столько, сколько строк оказалось в результате запроса *`category_sql`*.

Функция `crosstab` выдаёт одну выходную строку для каждой последовательной группы входных строк с одним значением `row_name`. Выходной столбец `row_name` плюс все «дополнительные» столбцы копируются из первой строки группы. Выходные столбцы значений заполняются содержимым полей `value` из строк с соответствующими значениями `category`. Если в поле `category` оказывается значение, отсутствующее в результате запроса *`category_sql`*, содержимое поля `value` в этой строке игнорируется. Выходные столбцы, для которых соответствующая категория не представлена ни в одной из входных строк группы, принимают значения NULL.

На практике в запросе *`source_sql`* всегда нужно указывать `ORDER BY 1`, чтобы все значения с одним `row_name` гарантированно выводились вместе. Порядок же категорий внутри группы не важен. Кроме того, важно, чтобы порядок значений, выдаваемых запросом *`category_sql`*, соответствовал заданному порядку выходных столбцов.

Два законченных примера:

```
create table sales(year int, month int, qty int);
insert into sales values(2007, 1, 1000);
insert into sales values(2007, 2, 1500);
insert into sales values(2007, 7, 500);
insert into sales values(2007, 11, 1500);
insert into sales values(2007, 12, 2000);
insert into sales values(2008, 1, 1000);

select * from crosstab(
  'select year, month, qty from sales order by 1',
  'select m from generate_series(1,12) m'
) as (
  year int,
  "Jan" int,
  "Feb" int,
  "Mar" int,
  "Apr" int,
  "May" int,
  "Jun" int,
  "Jul" int,
  "Aug" int,
  "Sep" int,
  "Oct" int,
  "Nov" int,
  "Dec" int
);
 year | Jan  | Feb  | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov  | Dec
------+------+------+-----+-----+-----+-----+-----+-----+-----+-----+------+------
 2007 | 1000 | 1500 |     |     |     |     | 500 |     |     |     | 1500 | 2000
 2008 | 1000 |      |     |     |     |     |     |     |     |     |      |
(2 rows)

CREATE TABLE cth(rowid text, rowdt timestamp, attribute text, val text);
```

```
INSERT INTO cth VALUES('test1','01 March 2003','temperature','42');
INSERT INTO cth VALUES('test1','01 March 2003','test_result','PASS');
INSERT INTO cth VALUES('test1','01 March 2003','volts','2.6987');
INSERT INTO cth VALUES('test2','02 March 2003','temperature','53');
INSERT INTO cth VALUES('test2','02 March 2003','test_result','FAIL');
INSERT INTO cth VALUES('test2','02 March 2003','test_startdate','01 March 2003');
INSERT INTO cth VALUES('test2','02 March 2003','volts','3.1234');

SELECT * FROM crosstab
(
  'SELECT rowid, rowdt, attribute, val FROM cth ORDER BY 1',
  'SELECT DISTINCT attribute FROM cth ORDER BY 1'
)
AS
(
       rowid text,
       rowdt timestamp,
       temperature int4,
       test_result text,
       test_startdate timestamp,
       volts float8
);
 rowid |          rowdt           | temperature | test_result |      test_startdate
  | volts
-------+--------------------------+-------------+-------------
+--------------------------+--------
 test1 | Sat Mar 01 00:00:00 2003 |          42 | PASS        |
  | 2.6987
 test2 | Sun Mar 02 00:00:00 2003 |          53 | FAIL        | Sat Mar 01 00:00:00
 2003 | 3.1234
(2 rows)
```

Вы можете создать предопределённые функции, чтобы не выписывать имена и типы результирующих столбцов в каждом запросе. Примеры приведены в предыдущем разделе. Нижележащая функция C для этой формы `crosstab` называется `crosstab_hash`.

F.54.1.5. connectby

```
connectby(text relname, text keyid_fld, text parent_keyid_fld
          [, text orderby_fld ], text start_with, int max_depth
          [, text branch_delim ])
```

Функция `connectby` выдаёт отображение данных, содержащихся в таблице, в иерархическом виде. Таблица должна содержать поле ключа, однозначно идентифицирующее строки, и поле ключа родителя, ссылающееся на родителя строки (если он есть). Функция `connectby` может вывести вложенное дерево, начиная с любой строки.

Параметры описаны в Таблице F.34.

Таблица F.34. Параметры connectby

Параметр	Описание
relname	Имя исходного отношения
keyid_fld	Имя поля ключа
parent_keyid_fld	Имя поля, содержащего ключ родителя
orderby_fld	Имя поля, по которому сортируются потомки (необязательно)
start_with	Значение ключа отправной строки

Параметр	**Описание**
max_depth	Максимальная глубина, на которую можно погрузиться, либо ноль для неограниченного погружения
branch_delim	Строка, разделяющая ключи в выводе ветви (необязательно)

Поля ключа и ключа родителя могут быть любого типа, но должны иметь общий тип. Заметьте, что значение *start_with* должно задаваться текстовой строкой, вне зависимости от типа поля ключа.

Функция connectby объявлена как возвращающая setof record, так что фактические имена и типы выходных столбцов должны определяться в предложении FROM вызывающего оператора SELECT, например так:

```
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0,
 '~')
    AS t(keyid text, parent_keyid text, level int, branch text, pos int);
```

Первые два выходных столбца используются для вывода ключа текущей строки и ключа её родителя; их тип должен соответствовать типу поля ключа. Третий выходной столбец задаёт глубину в дереве и должен иметь тип integer. Если передаётся параметр *branch_delim*, в следующем столбце выводятся ветви, и этот столбец должен иметь тип text. Наконец, если передаётся параметр *orderby_fld*, в последнем столбце выводятся последовательные числа, и он должен иметь тип integer.

В столбце «branch» показывается путь по ключам, приведший к текущей строке. Ключи разделяются заданной строкой *branch_delim*. Если выводить ветви не требуется, опустите параметр *branch_delim* и столбец branch в списке выходных столбцов.

Если порядок потомков одного родителя имеет значение, добавьте параметр *orderby_fld*, указывающий поле для упорядочивания потомков. Это поле может иметь любой тип, допускающий сортировку. Список выходных столбцов должен включать последним столбцом целочисленный столбец с последовательными значениями, если и только если передаётся параметр *orderby_fld*.

Параметры, представляющие имена таблицы и полей, копируются как есть в SQL-запросы, которые connectby генерирует внутри. Таким образом, их нужно заключить в двойные кавычки, если они содержат буквы в разном регистре или специальные символы. Также может понадобиться дополнить имя таблицы схемой.

С большими таблицами производительность будет неудовлетворительной, если не создать индекс по полю с ключом родителя.

Важно, чтобы строка *branch_delim* не фигурировала в значениях ключа, иначе connectby может некорректно сообщить об ошибке бесконечной вложенности. Заметьте, что если параметр *branch_delim* не задаётся, для выявления зацикленности применяется символ ~.

Пример:

```
CREATE TABLE connectby_tree(keyid text, parent_keyid text, pos int);

INSERT INTO connectby_tree VALUES('row1',NULL, 0);
INSERT INTO connectby_tree VALUES('row2','row1', 0);
INSERT INTO connectby_tree VALUES('row3','row1', 0);
INSERT INTO connectby_tree VALUES('row4','row2', 1);
INSERT INTO connectby_tree VALUES('row5','row2', 0);
INSERT INTO connectby_tree VALUES('row6','row4', 0);
INSERT INTO connectby_tree VALUES('row7','row3', 0);
INSERT INTO connectby_tree VALUES('row8','row6', 0);
INSERT INTO connectby_tree VALUES('row9','row5', 0);
```

```
-- с ветвями без orderby_fld (порядок результатов не гарантируется)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0, '~')
 AS t(keyid text, parent_keyid text, level int, branch text);
 keyid | parent_keyid | level |       branch
-------+--------------+-------+--------------------
 row2  |              |     0 | row2
 row4  | row2         |     1 | row2~row4
 row6  | row4         |     2 | row2~row4~row6
 row8  | row6         |     3 | row2~row4~row6~row8
 row5  | row2         |     1 | row2~row5
 row9  | row5         |     2 | row2~row5~row9
(6 rows)

-- без ветвей и без orderby_fld (порядок результатов не гарантируется)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0)
 AS t(keyid text, parent_keyid text, level int);
 keyid | parent_keyid | level
-------+--------------+-------
 row2  |              |     0
 row4  | row2         |     1
 row6  | row4         |     2
 row8  | row6         |     3
 row5  | row2         |     1
 row9  | row5         |     2
(6 rows)

-- с ветвями и с orderby_fld (заметьте, что row5 идёт перед row4)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0,
 '~')
 AS t(keyid text, parent_keyid text, level int, branch text, pos int);
 keyid | parent_keyid | level |       branch        | pos
-------+--------------+-------+---------------------+-----
 row2  |              |     0 | row2                |   1
 row5  | row2         |     1 | row2~row5           |   2
 row9  | row5         |     2 | row2~row5~row9      |   3
 row4  | row2         |     1 | row2~row4           |   4
 row6  | row4         |     2 | row2~row4~row6      |   5
 row8  | row6         |     3 | row2~row4~row6~row8 |   6
(6 rows)

-- без ветвей, с orderby_fld (заметьте, что row5 идёт перед row4)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0)
 AS t(keyid text, parent_keyid text, level int, pos int);
 keyid | parent_keyid | level | pos
-------+--------------+-------+-----
 row2  |              |     0 |   1
 row5  | row2         |     1 |   2
 row9  | row5         |     2 |   3
 row4  | row2         |     1 |   4
 row6  | row4         |     2 |   5
 row8  | row6         |     3 |   6
(6 rows)
```

F.54.2. Автор

Джо Конвей

F.55. tcn

Модуль `tcn` предлагает триггерную функцию, уведомляющую приёмники уведомлений об изменениях в любой таблице, к которой привязан триггер. Она должна использоваться в качестве триггера `AFTER` вида `FOR EACH ROW`.

Этой функции в операторе `CREATE TRIGGER` передаётся только один параметр, и он не является обязательным. Если этот параметр присутствует, он задаёт имя канала для уведомлений. В случае его отсутствия именем канала будет `tcn`.

Сообщение уведомления включает имя таблицы, букву, обозначающую тип выполняемой операции и пары имя столбца/значение для столбца первичного ключа. Каждая часть сообщения отделяется от следующей запятой. Для упрощения разбора сообщения регулярными выражениями имена таблицы и столбцов всегда заключаются в двойные кавычки, а значения данных — в апострофы. Внутренние кавычки и апострофы дублируются.

Далее приведена краткая демонстрация использования расширения.

```
test=# create table tcndata
test-#   (
test(#     a int not null,
test(#     b date not null,
test(#     c text,
test(#     primary key (a, b)
test(#   );
CREATE TABLE
test=# create trigger tcndata_tcn_trigger
test-#   after insert or update or delete on tcndata
test-#   for each row execute procedure triggered_change_notification();
CREATE TRIGGER
test=# listen tcn;
LISTEN
test=# insert into tcndata values (1, date '2012-12-22', 'one'),
test-#                            (1, date '2012-12-23', 'another'),
test-#                            (2, date '2012-12-23', 'two');
INSERT 0 3
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='1',"b"='2012-12-22'"
 received from server process with PID 22770.
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='1',"b"='2012-12-23'"
 received from server process with PID 22770.
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='2',"b"='2012-12-23'"
 received from server process with PID 22770.
test=# update tcndata set c = 'uno' where a = 1;
UPDATE 2
Asynchronous notification "tcn" with payload ""tcndata",U,"a"='1',"b"='2012-12-22'"
 received from server process with PID 22770.
Asynchronous notification "tcn" with payload ""tcndata",U,"a"='1',"b"='2012-12-23'"
 received from server process with PID 22770.
test=# delete from tcndata where a = 1 and b = date '2012-12-22';
DELETE 1
Asynchronous notification "tcn" with payload ""tcndata",D,"a"='1',"b"='2012-12-22'"
 received from server process with PID 22770.
```

F.56. test_decoding

Модуль `test_decoding` представляет пример модуля вывода логического декодирования. Он не делает ничего особенно полезного, но может послужить отправной точкой для разработки собственного модуля вывода.

Модуль `test_decoding` получает WAL через механизм логического декодирования и переводит его в текстовое представление выполняемых операций.

Типичный вывод этого модуля, работающего через интерфейс логического декодирования SQL, может выглядеть так:

```
postgres=# SELECT * FROM pg_logical_slot_get_changes('test_slot', NULL, NULL, 'include-
xids', '0');
   lsn     | xid |                       data
-----------+-----+-------------------------------------------------
 0/16D30F8 | 691 | BEGIN
 0/16D32A0 | 691 | table public.data: INSERT: id[int4]:2 data[text]:'arg'
 0/16D32A0 | 691 | table public.data: INSERT: id[int4]:3 data[text]:'demo'
 0/16D32A0 | 691 | COMMIT
 0/16D32D8 | 692 | BEGIN
 0/16D3398 | 692 | table public.data: DELETE: id[int4]:2
 0/16D3398 | 692 | table public.data: DELETE: id[int4]:3
 0/16D3398 | 692 | COMMIT
(8 rows)
```

F.57. tsm_system_rows

Модуль `tsm_system_rows` предоставляет метод извлечения выборки `SYSTEM_ROWS`, который можно использовать в предложении `TABLESAMPLE` команды SELECT.

Этот метод извлечения выборки принимает один целочисленный аргумент, задающий максимальное число выбираемых строк. Результирующая выборка будет содержать в точности столько строк, если только в таблице не оказывается меньше заданного числа строк (в этом случае выдаётся вся таблица).

Как и встроенный метод извлечения выборки `SYSTEM`, `SYSTEM_ROWS` производит выборку на уровне блоков, так что выборка будет не полностью случайной, а может подвергаться эффектам кластеризации, особенно когда запрашивается небольшое число строк.

`SYSTEM_ROWS` не поддерживает предложение `REPEATABLE`.

F.57.1. Примеры

Пример получения выборки из таблицы с применением метода `SYSTEM_ROWS`. Сначала нужно установить расширение:

```
CREATE EXTENSION tsm_system_rows;
```

Затем вы можете использовать его в команде `SELECT`, например так:

```
SELECT * FROM my_table TABLESAMPLE SYSTEM_ROWS(100);
```

Эта команда выдаст выборку из 100 строк из таблицы `my_table` (а если в таблице не окажется 100 видимых строк, будут возвращены все строки).

F.58. tsm_system_time

Модуль `tsm_system_time` предоставляет метод извлечения выборки `SYSTEM_TIME`, который можно использовать в предложении `TABLESAMPLE` команды SELECT.

Этот метод извлечения выборки принимает в единственном аргументе число с плавающей точкой, задающее максимальное время (в миллисекундах), которое отводится на чтение таблицы. Это даёт возможность непосредственно управлять длительностью выполнения запроса, ценой того, что размер выборки оказывается трудно предсказуемым. Результирующая выборка будет содержать столько строк, сколько удастся прочитать за отведённое время, если только быстрее не будет прочитана вся таблица.

Как и встроенный метод извлечения выборки `SYSTEM`, `SYSTEM_TIME` производит выборку на уровне блоков, так что выборка будет не полностью случайной, а может подвергаться эффектам кластеризации, особенно когда выбирается небольшое число строк.

SYSTEM_TIME не поддерживает предложение REPEATABLE.

F.58.1. Примеры

Пример получения выборки из таблицы с применением метода SYSTEM_TIME. Сначала нужно установить расширение:

```
CREATE EXTENSION tsm_system_time;
```

Затем вы можете использовать его в команде SELECT, например так:

```
SELECT * FROM my_table TABLESAMPLE SYSTEM_TIME(1000);
```

Эта команда выдаст настолько большую выборку из my_table, насколько много строк она успеет прочитать за 1 секунду (1000 миллисекунд). Разумеется, если за 1 секунду удастся прочитать всю таблицу, будут возвращены все её строки.

F.59. unaccent

Модуль unaccent представляет словарь текстового поиска, который убирает надстрочные (диакритические) знаки из лексем. Это фильтрующий словарь, что значит, что выводимые им данные всегда передаются следующему словарю (если он есть), в отличие от нормальных словарей. Применяя его, можно выполнить полнотекстовый поиск без учёта ударений (диакритики).

Текущую реализацию unaccent нельзя использовать в качестве нормализующего словаря для словаря thesaurus.

F.59.1. Конфигурирование

Словарь unaccent принимает следующие параметры:

- Параметр RULES задаёт базовое имя файла со списком правил преобразования. Этот файл должен находиться в каталоге $SHAREDIR/tsearch_data/ (где под $SHAREDIR понимается каталог с общими данными инсталляции Postgres Pro). Имя файла должно заканчиваться расширением .rules (которое не нужно указывать в параметре RULES).

Файл правил имеет следующий формат:

- Каждая строка представляет одно правило перевода, состоящее из символа с диакритикой, за которым следует символ без диакритики. Первый символ переводится во второй. Например,

  ```
  À       A
  Á       A
  Â       A
  Ã       A
  Ä       A
  Å       A
  Æ       AE
  ```

  Эти два символа должны разделяться пробельным символом, а любые начальные или конечные пробельные символы в этой строке игнорируются.

- В качестве альтернативного варианта, если в строке задан всего один символ, вхождения этого символа будут удаляться; это полезно для языков, в которых диакритика представляется отдельными символами.

- На самом деле роль «символа» может играть любая строка, не содержащая пробельные символы, так что словари unaccent могут быть полезны и для другого рода замены подстрок, а не только для удаления диакритик.

- Как и другие файлы конфигурации текстового поиска в Postgres Pro, файл правил должен иметь кодировку UTF-8. При загрузке данные из него будут автоматически преобразованы

в кодировку текущей базы данных. Все строки в нём, содержащие непереводимые символы, просто игнорируются, так что файлы правил могут содержать правила, неприменимые в текущей кодировке.

Более полный набор правил, непосредственно пригодный для большинства европейских языков, можно найти в файле `unaccent.rules`, который помещается в `$SHAREDIR/tsearch_data/`, когда устанавливается модуль `unaccent`. Этот файл правил переводит буквы с диакритикой в те же буквы без диакритики, а также разворачивает лигатуры в равнозначные последовательности простых символов (например, Æ в AE).

F.59.2. Использование

При установке расширения `unaccent` в базе создаётся шаблон текстового поиска `unaccent` и словарь `unaccent` на его основе. Для словаря `unaccent` по умолчанию определяется параметр `RULES='unaccent'`, благодаря чему его можно сразу использовать со стандартным файлом `unaccent.rules`. При желании вы можете изменить этот параметр, например, так

```
mydb=# ALTER TEXT SEARCH DICTIONARY unaccent (RULES='my_rules');
```

или создать новые словари на основе этого шаблона.

Протестировать этот словарь можно так:

```
mydb=# select ts_lexize('unaccent','Hôtel');
 ts_lexize
-----------
 {Hotel}
(1 row)
```

Следующий пример показывает, как вставить словарь `unaccent` в конфигурацию текстового поиска:

```
mydb=# CREATE TEXT SEARCH CONFIGURATION fr ( COPY = french );
mydb=# ALTER TEXT SEARCH CONFIGURATION fr
        ALTER MAPPING FOR hword, hword_part, word
        WITH unaccent, french_stem;
mydb=# select to_tsvector('fr','Hôtels de la Mer');
    to_tsvector
-------------------
 'hotel':1 'mer':4
(1 row)

mydb=# select to_tsvector('fr','Hôtel de la Mer') @@ to_tsquery('fr','Hotels');
 ?column?
----------
 t
(1 row)

mydb=# select ts_headline('fr','Hôtel de la Mer',to_tsquery('fr','Hotels'));
      ts_headline
------------------------
 <b>Hôtel</b> de la Mer
(1 row)
```

F.59.3. Функции

Функция `unaccent()` удаляет надстрочные (диакритические) знаки из заданной строки. По сути она представляет собой обёртку вокруг словарей в стиле `unaccent`, но может вызываться и вне обычного контекста текстового поиска.

```
unaccent([словарь regdictionary,] строка text) returns text
```

Если аргумент *словарь* опущен, будет использоваться словарь с именем `unaccent`, находящийся в той же схеме, что и сама функция `unaccent()`.

Пример:

```
SELECT unaccent('unaccent', 'Hôtel');
SELECT unaccent('Hôtel');
```

F.60. uuid-ossp

Модуль `uuid-ossp` предоставляет функции для генерирования универсальных уникальных идентификаторов (UUID) по одному из нескольких стандартных алгоритмов. В нём также есть функции, выдающие специальные UUID-константы.

F.60.1. Функции `uuid-ossp`

В Таблице F.35 показаны функции, предназначенные для генерации UUID. Четыре алгоритма для генерации UUID, обозначаемые номерами версий 1, 3, 4 и 5, описаны в стандартах ITU-T Rec. X.667, ISO/IEC 9834-8:2005 и RFC 4122. (Алгоритма версии 2 нет.) Каждый из этих алгоритмов предназначен для различных сфер применения.

Таблица F.35. Функции для генерирования UUID

Функция	Описание
`uuid_generate_v1()`	Эта функция генерирует UUID версии 1. Такой UUID включает в себя MAC-адрес компьютера и текущее время. Заметьте, что UUID такого типа раскрывают «личность» компьютера, создавшего идентификатор, и время этой операции, что может быть неприемлемым для определённых приложений, где важна конфиденциальность.
`uuid_generate_v1mc()`	Эта функция генерирует UUID версии 1, но вместо реального MAC-адреса компьютера используется случайный групповой MAC-адрес.
`uuid_generate_v3(namespace uuid, name text)`	Эта функция генерирует UUID версии 3 для заданного пространства имён UUID и указанного имени. Пространство имён должно задаваться одной из специальных констант, которые выдаются функциями `uuid_ns_*()`, перечисленными в Таблице F.36. (Хотя теоретически это может быть любой UUID.) Имя задаёт идентификатор в выбранном пространстве имён. Например: `SELECT uuid_generate_v3(uuid_ns_url(), 'http://www.postgresql.org');` Из параметра name будет получен MD5-хеш, так что из сгенерированного UUID нельзя будет восстановить имя. В генерируемых таким алгоритмом UUID нет элемента случайности или зависимости от окружения, так что они могут быть воспроизведены.
`uuid_generate_v4()`	Эта функция генерирует UUID версии 4, который всецело определяется случайными числами.

Функция	Описание
`uuid_generate_v5(namespace uuid, name text)`	Эта функция генерирует UUID версии 5, который похож на версию 3, но хеш рассчитывается по алгоритму SHA-1. Версия 5 предпочтительнее версии 3, так как SHA-1 считается более безопасным, чем MD5.

Таблица F.36. Функции, возвращающие UUID-константы

`uuid_nil()`	«Нулевой» UUID, который не считается действительным UUID.
`uuid_ns_dns()`	Константа, обозначающая пространство имён DNS для UUID.
`uuid_ns_url()`	Константа, обозначающая пространство имён URL для UUID.
`uuid_ns_oid()`	Константа, обозначающая пространство имён идентификаторов объектов ISO (OID, ISO Object Identifier) для UUID. (Здесь имеются в виду идентификаторы объектов ASN.1, которые никак не связаны с OID, применяемыми в Postgres Pro.)
`uuid_ns_x500()`	Константа, обозначающая пространство имён с уникальными именами X.500 для UUID.

F.60.2. Сборка `uuid-ossp`

В прошлом этот модуль зависел от библиотеки OSSP UUID, что отразилось в его имени. Хотя библиотеку OSSP UUID всё ещё можно найти по адресу *http://www.ossp.org/pkg/lib/uuid/*, она плохо поддерживается и её становится всё сложнее портировать на новые платформы. Поэтому модуль `uuid-ossp` теперь на некоторых платформах можно собирать без библиотеки OSSP. Во FreeBSD и некоторых других ОС на базе BSD подходящие функции формирования UUID включены в системную библиотеку `libc`. В Linux, macOS и некоторых других платформах подходящие функции предоставляются библиотекой `libuuid`, которая изначально пришла из проекта `e2fsprogs` (хотя в современных дистрибутивах Linux она является частью пакета `util-linux-ng`). Вызывая `configure`, передайте ключ `--with-uuid=bsd`, чтобы использовать функции BSD, либо `--with-uuid=e2fs`, чтобы использовать `libuuid` из `e2fsprogs`, либо ключ `--with-uuid=ossp`, чтобы использовать библиотеку OSSP UUID. В конкретной системе может быть установлено сразу несколько библиотек, поэтому `configure` не выбирает библиотеку автоматически.

> **Примечание**
>
> Если вам нужны только случайные UUID (версии 4), в качестве альтернативы вы можете использовать функцию `gen_random_uuid()` из модуля pgcrypto.

F.60.3. Автор

Питер Эйзентраут `<peter_e@gmx.net>`

F.61. xml2

Модуль `xml2` предоставляет функции для выполнения запросов XPath и преобразований XSLT.

F.61.1. Уведомление об актуальности

Начиная с PostgreSQL 8.3, функциональность, связанная с XML, основана на стандарте SQL/XML и включена в ядро сервера. Эта функциональность охватывает проверку синтаксиса

XML и запросы XPath, что в частности делает и этот модуль, но он имеет абсолютно несовместимый API. Этот модуль планируется удалить в будущей версии Postgres Pro в пользу нового стандартного API, так что мы рекомендуем вам попробовать перевести свои приложения на новый API. Если вы обнаружите, что какая-то функциональность этого модуля не представлена новым API в подходящей форме, пожалуйста, напишите о вашем затруднении в <pgsql-hackers@lists.postgresql.org>, чтобы этот недостаток был рассмотрен и, возможно, устранён.

F.61.2. Описание функций

Функции, предоставляемые этим модулем, перечислены в Таблице F.37. Эти функции позволяют выполнять простой разбор XML и запросы XPath. Все их аргументы имеют тип `text`, поэтому для краткости типы опущены.

Таблица F.37. Функции

Функция	Возвращает	Описание
`xml_valid(document)`	`bool`	Эта функция разбирает текст документа, переданный в параметре, и возвращает true, если это правильно сформированный XML. (Замечание: это псевдоним стандартной функции Postgres Pro `xml_is_well_formed()` . Имя `xml_valid()` технически некорректно, так как понятия правильности формата (well-formed) и допустимости (valid) в XML различаются.)
`xpath_string(document, query)`	`text`	Эти функции обрабатывают запрос XPath для переданного документа и приводят результат к указанному типу.
`xpath_number(document, query)`	`float4`	
`xpath_bool(document, query)`	`bool`	
`xpath_nodeset(document, query, toptag, itemtag)`	`text`	Эта функция обрабатывает запрос для документа и помещает результат внутрь XML-тегов. Если результат содержит несколько значений, она выдаст: `<toptag>` `<itemtag>Значение 1, которое может быть фрагментом XML</itemtag>` `<itemtag>Значение 2....</itemtag>` `</toptag>` Если `toptag` или `itemtag` — пустая строка, соответствующий тег опускается.
`xpath_nodeset(document, query)`	`text`	Подобна `xpath_nodeset(document, query, toptag,`

Функция	Возвращает	Описание
		`itemtag)`, но выводит результат без обоих тегов.
`xpath_nodeset(document, query, itemtag)`	`text`	Подобна `xpath_nodeset(document, query, toptag, itemtag)`, но выводит результат без `toptag`.
`xpath_list(document, query, separator)`	`text`	Эта функция возвращает несколько значений, вставляя между ними заданный разделитель, например: `Значение 1,Значение 2, Значение 3`, если разделитель — знак `,`.
`xpath_list(document, query)`	`text`	Это обёртка предыдущей функции, устанавливающая в качестве разделителя знак `,`.

F.61.3. xpath_table

```
xpath_table(text key, text document, text relation, text xpaths, text criteria) returns
 setof record
```

Табличная функция `xpath_table` выполняет набор запросов XPath для каждого из набора документов и возвращает результаты в виде таблицы. В первом столбце результата возвращается первичный ключ из таблицы документов, так что результат оказывается готовым к применению в соединениях. Параметры функции описаны в Таблице F.38.

Таблица F.38. Параметры `xpath_table`

Параметр	Описание
key	имя «ключевого» поля — содержимое этого поля просто окажется в первом столбце выходной таблицы, то есть оно указывает на запись, из которой была получена определённая выходная строка (см. замечание о нескольких значениях ниже)
document	имя поля, содержащего XML-документ
relation	имя таблицы (или представления), содержащей документы
xpaths	одно или несколько выражений XPath, разделённых символом `|`
criteria	содержимое предложения WHERE. Оно не может быть пустым, так что если вам нужно обработать все строки в отношении, напишите `true` или `1=1`

Эти параметры (за исключением строк XPath) просто подставляются в обычный оператор SQL SELECT, так что у вас есть определённая гибкость — оператор выглядит так:

```
SELECT <key>, <document> FROM <relation> WHERE <criteria>
```

поэтому в этих параметрах можно передать *всё*, что будет корректно воспринято в этих позициях. Этот SELECT должен возвращать ровно два столбца (что и будет иметь место, если только вы не перечислите несколько полей в параметрах key или document). Будьте осторожны — при таком

примитивном подходе обязательно нужно проверять все значения, получаемые от пользователя, во избежание атак с инъекцией SQL.

Эта функция предназначена для использования в выражении FROM, с предложением AS, задающим выходные столбцы; например:

```
SELECT * FROM
xpath_table('article_id',
            'article_xml',
            'articles',
            '/article/author|/article/pages|/article/title',
            'date_entered > ''2003-01-01'' ')
AS t(article_id integer, author text, page_count integer, title text);
```

Предложение AS определяет имена и типы столбцов в выходной таблице. Первым определяется «ключевое» поле, а за ним поля, соответствующие запросам XPath. Если запросов XPath больше, чем столбцов в результате, лишние запросы будут игнорироваться. Если же результирующих столбцов больше, чем запросов XPath, дополнительные столбцы принимают значение NULL.

Заметьте, что в этом примере столбец результата page_count определён как целочисленный. Данная функция внутри имеет дело со строковыми значениями, так что, когда вы указываете, что в результате нужно получить целое число, она берёт текстовое представление результата XPath и, применяя функции ввода Postgres Pro, преобразует её в целое число (или в тот тип, который указан в предложении AS). Если она не сможет сделать это, произойдёт ошибка — например, если результат пустой — так что если вы допускаете возможность таких проблем с данными, возможно, будет лучше просто оставить для столбца тип text.

Вызывающий оператор SELECT не обязательно должен быть простым SELECT * — он может обращаться к выходным столбцам по именам и соединять их с другими таблицами. Эта функция формирует виртуальную таблицу, с которой вы можете выполнять любые операции, какие пожелаете (например, агрегировать, соединять, сортировать данные и т. д.). Поэтому возможен и такой запрос:

```
SELECT t.title, p.fullname, p.email
FROM xpath_table('article_id', 'article_xml', 'articles',
                 '/article/title|/article/author/@id',
                 'xpath_string(article_xml,''/article/@date'') > ''2003-03-20'' ')
       AS t(article_id integer, title text, author_id integer),
     tblPeopleInfo AS p
WHERE t.author_id = p.person_id;
```

в качестве более сложного примера. Разумеется, для удобства вы можете завернуть весь этот запрос в представление.

F.61.3.1. Результаты с набором значений

Функция xpath_table рассчитана на то, что результатом каждого запроса XPath может быть набор данных, так что количество возвращённых этой функцией строк может не совпадать с количеством входных документов. В первой строке возвращается первый результат каждого запроса, во второй — второй результат и т. д. Если один из запросов возвращает меньше значений, чем другие, вместо недостающих значений будет возвращаться NULL.

В некоторых случаях пользователь знает, что некоторый запрос XPath будет возвращать только один результат (возможно, уникальный идентификатор документа) — если он используется рядом с запросом XPath, возвращающим несколько результатов, результат с одним значением будет выведен только в первой выходной строке. Чтобы исправить это, можно воспользоваться полем ключа и соединить результат с более простым запросом XPath. Например:

```
CREATE TABLE test (
    id int PRIMARY KEY,
    xml text
);
```

```
INSERT INTO test VALUES (1, '<doc num="C1">
<line num="L1"><a>1</a><b>2</b><c>3</c></line>
<line num="L2"><a>11</a><b>22</b><c>33</c></line>
</doc>');

INSERT INTO test VALUES (2, '<doc num="C2">
<line num="L1"><a>111</a><b>222</b><c>333</c></line>
<line num="L2"><a>111</a><b>222</b><c>333</c></line>
</doc>');

SELECT * FROM
  xpath_table('id','xml','test',
              '/doc/@num|/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
              'true')
  AS t(id int, doc_num varchar(10), line_num varchar(10), val1 int, val2 int, val3 int)
WHERE id = 1 ORDER BY doc_num, line_num

 id | doc_num | line_num | val1 | val2 | val3
----+---------+----------+------+------+------
  1 | C1      | L1       |    1 |    2 |    3
  1 |         | L2       |   11 |   22 |   33
```

Чтобы получить `doc_num` в каждой строке, можно вызывать `xpath_table` дважды и соединить результаты:

```
SELECT t.*,i.doc_num FROM
  xpath_table('id', 'xml', 'test',
              '/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
              'true')
    AS t(id int, line_num varchar(10), val1 int, val2 int, val3 int),
  xpath_table('id', 'xml', 'test', '/doc/@num', 'true')
    AS i(id int, doc_num varchar(10))
WHERE i.id=t.id AND i.id=1
ORDER BY doc_num, line_num;

 id | line_num | val1 | val2 | val3 | doc_num
----+----------+------+------+------+---------
  1 | L1       |    1 |    2 |    3 | C1
  1 | L2       |   11 |   22 |   33 | C1
(2 rows)
```

F.61.4. Функции XSLT

Если установлена libxslt, доступны следующие функции:

F.61.4.1. `xslt_process`

```
xslt_process(text document, text stylesheet, text paramlist) returns text
```

Эта функция применяет стиль XSL к документу и возвращает результат преобразования. В `paramlist` передаётся список присваиваний значений параметрам, которые будут использоваться в преобразовании, в форме `a=1,b=2`. Учтите, что разбор параметров выполнен очень просто: значения параметров не могут содержать запятые!

Есть также версия `xslt_process` с двумя аргументами, которая не передаёт никакие параметры преобразованию.

F.61.5. Автор

Джон Грей `<jgray@azuli.co.uk>`

Разработку этого модуля спонсировала компания Torchbox Ltd. (www.torchbox.com). Этот модуль выпускается под той же лицензией BSD, что и Postgres Pro.

Приложение G. Дополнительно поставляемые программы

В этом и предыдущем приложении содержится информация о дополнительных модулях, поставляемых с дистрибутивом Postgres Pro Standard. За дополнительной информацией о серверных расширениях и подключаемых компонентах обратитесь к Приложению F.

В этом приложении описываются вспомогательные программы. После установки их можно найти в подкаталоге `bin` инсталляции Postgres Pro Standard и использовать как любую другую программу.

G.1. Клиентские приложения

В этом разделе рассматриваются клиентские приложения Postgres Pro Standard. Их можно запускать откуда угодно, независимо от того, где находится сервер баз данных. Обратитесь также к Справке: «Клиентские приложения Postgres Pro» за информацией о клиентских приложениях, относящихся к ядру дистрибутива Postgres Pro Standard.

oid2name

oid2name — преобразовать в имена OID и номера файловых узлов в каталоге данных Postgres Pro

Синтаксис

`oid2name` [*параметр*...]

Описание

oid2name — вспомогательная программа, помогающая администраторам исследовать структуру файлов Postgres Pro. Чтобы использовать её, необходимо понимать структуру базы данных на уровне файлов, описанную в Главе 63.

> **Примечание**
>
> Имя «oid2name» сложилось исторически и на самом деле вводит в заблуждение, так как чаще всего, используя её, вы будете иметь дело с номерами файловых узлов таблиц (эти номера образуют имена файлов в каталоге баз данных). Необходимо понимать, что OID таблиц отличаются от номеров файловых узлов!

Программа oid2name подключается к целевой базе данных и извлекает информацию об OID, файловых узлах и/или именах таблиц. С её помощью можно также просмотреть OID базы данных или табличного пространства.

Параметры

oid2name принимает следующие аргументы командной строки:

`-f` *файловый_узел*
: показать информацию о таблице, к которой относится *файловый_узел*

`-i`
: включить в вывод индексы и последовательности

`-o` *oid*
: показать информацию о таблице с OID, равным *oid*

`-q`
: не выводить заголовки (полезно для скриптов)

`-s`
: показать OID табличных пространств

`-S`
: включить в вывод системные объекты (те, что находятся в схемах `information_schema`, `pg_toast` и `pg_catalog`)

`-t` *шаблон_имён_таблиц*
: показать информацию о таблицах, подпадающих под *шаблон_имён_таблиц*

`-V`
`--version`
: вывести версию oid2name и завершиться.

`-x`
: вывести дополнительные сведения о каждом показываемом объекте: имя табличного пространства, имя схемы и OID

`-?`
`--help`

вывести справку об аргументах командной строки oid2name и завершиться.

oid2name также принимает в командной строке следующие аргументы, задающие параметры подключения:

`-d` *`база_данных`*

целевая база данных

`-H` *`сервер`*

адрес сервера баз данных

`-p` *`порт`*

порт сервера баз данных

`-U` *`имя_пользователя`*

имя пользователя для подключения

`-P` *`пароль`*

пароль (устаревшая возможность — указание пароля в командной строке угрожает безопасности)

Чтобы получить информацию об определённых таблицах, выберите их с аргументом `-o`, `-f` и/или `-t`. Параметр `-o` принимает OID, `-f` — файловый узел, а `-t` — имя таблицы (на самом деле он принимает шаблон `LIKE`, так что в нём можно задать что-то вроде `foo%`). Вы можете использовать эти аргументы в любом количестве; в вывод будут включены все объекты, соответствующие любым из этих указаний. Но заметьте, что эти аргументы будут выбирать только объекты в базе данных, заданной ключом `-d`.

Если аргументы `-o`, `-f` и `-t` отсутствуют, но передаётся `-d`, будут выведены все таблицы в базе данных с именем, заданным в `-d`. В этом режиме набором выводимых данных управляют параметры `-S` и `-i`.

Если отсутствует и аргумент `-d`, выводится список OID баз данных. Также можно получить список табличных пространств, передав аргумент `-s`.

Замечания

Программе oid2name требуется работающий сервер баз данных с исправными системными каталогами. Таким образом, в ситуациях с катастрофическими повреждениями базы данных её использование ограничено.

Примеры

```
$ # Что вообще есть на этом сервере базы данных?
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  17228       alvherre  pg_default
  17255     regression  pg_default
  17227      template0  pg_default
      1      template1  pg_default

$ oid2name -s
All tablespaces:
     Oid  Tablespace Name
-------------------------
    1663       pg_default
```

```
    1664        pg_global
  155151         fastdisk
  155152          bigdisk

$ # Хорошо, давайте взглянем на базу alvherre
$ cd $PGDATA/base/17228

$ # Получим первые 10 объектов базы (отсортированных по размеру) в табличном
 пространстве по умолчанию
$ ls -lS * | head -10
-rw-------  1 alvherre alvherre 136536064 sep 14 09:51 155173
-rw-------  1 alvherre alvherre  17965056 sep 14 09:51 1155291
-rw-------  1 alvherre alvherre   1204224 sep 14 09:51 16717
-rw-------  1 alvherre alvherre    581632 sep  6 17:51 1255
-rw-------  1 alvherre alvherre    237568 sep 14 09:50 16674
-rw-------  1 alvherre alvherre    212992 sep 14 09:51 1249
-rw-------  1 alvherre alvherre    204800 sep 14 09:51 16684
-rw-------  1 alvherre alvherre    196608 sep 14 09:50 16700
-rw-------  1 alvherre alvherre    163840 sep 14 09:50 16699
-rw-------  1 alvherre alvherre    122880 sep  6 17:51 16751

$ # Поинтересуемся, что за файл 155173...
$ oid2name -d alvherre -f 155173
From database "alvherre":
  Filenode  Table Name
----------------------
    155173    accounts

$ # Можно узнать сразу о нескольких объектах
$ oid2name -d alvherre -f 155173 -f 1155291
From database "alvherre":
  Filenode     Table Name
-------------------------
    155173       accounts
   1155291  accounts_pkey

$ # Можно добавить другие параметры и получить дополнительные подробности с -x
$ oid2name -d alvherre -t accounts -f 1155291 -x
From database "alvherre":
  Filenode     Table Name      Oid  Schema  Tablespace
-----------------------------------------------------
    155173       accounts   155173  public  pg_default
   1155291  accounts_pkey  1155291  public  pg_default

$ # Вычислить объём, который занимает на диске каждый объект БД
$ du [0-9]* |
> while read SIZE FILENODE
> do
>   echo "$SIZE       `oid2name -q -d alvherre -i -f $FILENODE`"
> done
16            1155287  branches_pkey
16            1155289  tellers_pkey
17561            1155291  accounts_pkey
...

$ # То же самое, но с сортировкой по размеру
$ du [0-9]* | sort -rn | while read SIZE FN
> do
```

```
>   echo "$SIZE   `oid2name -q -d alvherre -f $FN`"
> done
133466             155173    accounts
17561             1155291  accounts_pkey
1177               16717  pg_proc_proname_args_nsp_index
...

$ # Просмотреть содержимое табличных пространств можно в каталоге pg_tblspc
$ cd $PGDATA/pg_tblspc
$ oid2name -s
All tablespaces:
     Oid  Tablespace Name
------------------------
    1663       pg_default
    1664        pg_global
  155151         fastdisk
  155152          bigdisk

$ # Объекты каких баз данных находятся в табличном пространстве "fastdisk"?
$ ls -d 155151/*
155151/17228/  155151/PG_VERSION

$ # И что это за база данных 17228?
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  17228       alvherre  pg_default
  17255     regression  pg_default
  17227      template0  pg_default
      1      template1  pg_default

$ # Давайте посмотрим, какие объекты этой базы содержатся в данном табличном
 пространстве.
$ cd 155151/17228
$ ls -l
total 0
-rw-------  1 postgres postgres 0 sep 13 23:20 155156

$ # Мда, таблица невелика... и что это за таблица?
$ oid2name -d alvherre -f 155156
From database "alvherre":
  Filenode  Table Name
----------------------
    155156         foo
```

Автор

Б. Палмер <bpalmer@crimelabs.net>

pg_probackup

pg_probackup — управление резервным копированием и восстановлением кластеров баз данных Postgres Pro

Синтаксис

pg_probackup version

pg_probackup help [*команда*]

pg_probackup init -B *каталог_копий*

pg_probackup add-instance -B *каталог_копий* -D *каталог_данных* --instance *имя_экземпляра*

pg_probackup del-instance -B *каталог_копий* --instance *имя_экземпляра*

pg_probackup set-config -B *каталог_копий* --instance *имя_экземпляра* [*параметр*...]

pg_probackup set-backup -B *каталог_копий* --instance *имя_экземпляра* -i *ид_резервной_копии* [*параметр*...]

pg_probackup show-config -B *каталог_копий* --instance *имя_экземпляра* [--format=*формат*]

pg_probackup show -B *каталог_копий* [*параметр*...]

pg_probackup backup -B *каталог_копий* --instance *имя_экземпляра* -b *режим_копирования* [*параметр*...]

pg_probackup restore -B *каталог_копий* --instance *имя_экземпляра* [*параметр*...]

pg_probackup checkdb -B *каталог_копий* --instance *имя_экземпляра* -D *каталог_данных* [параметр...]

pg_probackup validate -B *каталог_копий* [*параметр*...]

pg_probackup merge -B *каталог_копий* --instance *имя_экземпляра* -i *ид_резервной_копии* [*параметр*...]

pg_probackup delete -B *каталог_копий* --instance *имя_экземпляра* { -i *ид_резервной_копии* | --delete-wal | --delete-expired | --merge-expired } [*параметр*...]

pg_probackup archive-push -B *каталог_копий* --instance *имя_экземпляра* --wal-file-path *путь_файлов_wal* --wal-file-name *имя_файла_wal* [*параметр*...]

pg_probackup archive-get -B *каталог_копий* --instance *имя_экземпляра* --wal-file-path *путь_файлов_wal* --wal-file-name *имя_файла_wal* [*параметр*...]

pg_probackup catchup -b *режим_синхронизации* --source-pgdata=*путь_к_копируемому_каталогу_данных* --destination-pgdata=*путь_к_целевому_каталогу_данных* [*параметр*...]

Описание

pg_probackup — это утилита для управления резервным копированием и восстановлением кластеров баз данных Postgres Pro. Она предназначена для регулярного создания резервных копий экземпляра Postgres Pro, позволяющих восстанавливать сервер в случае необходимости. pg_probackup поддерживает PostgreSQL версии 9.5 и новее.

- Обзор
- Установка и подготовка

- Справка по командной строке
- Использование

Обзор

По сравнению с другими средствами резервного копирования pg_probackup имеет следующие преимущества, полезные для реализации различных стратегий резервного копирования и работы с базами данных большого объёма:

- Инкрементальное копирование: выбирая один из трёх режимов инкрементального копирования, вы можете реализовать стратегию резервного копирования, соответствующую вашему профилю транзакционной нагрузки. Это позволяет сэкономить место на диске и создавать копии быстрее, чем при полном копировании. Восстановление инкрементальных копий также осуществляется быстрее, чем воспроизведение файлов WAL.
- Инкрементальное восстановление: ускорение восстановления из копии благодаря повторному использованию неизменённых страниц, имеющихся в `PGDATA`.
- Проверка: автоматический контроль целостности данных и проверка резервных копий без восстановления данных кластера.
- Контроль целостности: выполняемая по запросу проверка экземпляра Postgres Pro с помощью команды `checkdb`.
- Политика хранения: управление архивами WAL и резервными копиями в соответствии с установленными правилами их хранения. Вы можете ограничить хранение резервных копий по времени или их количеству, а также переопределить время жизни (TTL) для избранных копий. Потерявшие актуальность резервные копии могут объединяться или удаляться.
- Параллельное выполнение: выполнение внутренних процессов команд `backup`, `restore`, `merge`, `delete`, `validate` и `checkdb` в несколько параллельных потоков.
- Сжатие: хранение копируемых данных в сжатом состоянии для экономии дискового пространства.
- Исключение дублирования: экономия дискового пространства за счёт фильтрации при инкрементальном копировании файлов, не содержащих непосредственно данные (например, файлов `_vm` или `_fsm`), если эти файлы не изменялись с момента создания предыдущей копии в цепочке инкрементальных копий.
- Удалённый режим работы: выполнение резервного копирования экземпляра Postgres Pro, находящегося в удалённой системе, и удалённое восстановление.
- Получение резервной копии с ведомого: исключение дополнительной нагрузки на ведущий сервер.
- Архивирование внешних каталогов: резервное копирование файлов и каталогов, расположенных вне каталога данных Postgres Pro (`PGDATA`), например скриптов, файлов конфигурации, журналов или SQL-дампов.
- Каталогизация резервных копий: получение списка резервных копий и соответствующей метаинформации в виде простого текста или JSON.
- Каталогизация архивов WAL: получение списка всех линий времени в WAL и соответствующей метаинформации в виде простого текста или JSON.
- Частичное восстановление: восстановление только избранных баз данных.
- Синхронизация: клонирование экземпляра Postgres Pro, в результате которого отставший ведомый сервер синхронизируется с ведущим.

Для управления резервными копиями pg_probackup создаёт *каталог резервных копий*. В этом каталоге сохраняются все файлы резервных копий с дополнительной метаинформацией, а также архивы WAL, необходимые для восстановления на момент времени. Вы можете хранить резервные копии разных экземпляров в отдельных подкаталогах одного каталога копий.

Используя pg_probackup, вы можете выполнять полное или инкрементальное резервное копирование:

- Полные резервные копии содержат все файлы данных, необходимые для восстановления кластера баз данных с нуля.
- Инкрементальные копии создаются на уровне страниц и включают только те данные, которые изменились со времени последнего копирования. Это позволяет сэкономить место на диске

и создавать копии быстрее, чем при полном копировании. Восстановление инкрементальных копий также осуществляется быстрее, чем воспроизведение файлов WAL. pg_probackup поддерживает следующие режимы инкрементального копирования:

- Разностное копирование. В режиме `DELTA` pg_probackup считывает все файлы данных в каталоге данных и копирует только те страницы, которые изменились со времени предыдущего копирования. Для использования данного режима не требуется производить непрерывное архивирование. В этом режиме объём ввода/вывода может равняться объёму при полном резервном копировании.
- Страничное копирование. В режиме `PAGE` pg_probackup сканирует все файлы WAL в архиве с момента создания предыдущей полной или инкрементальной копии. Новая резервная копия будет содержать только страницы, фигурирующие в записях WAL. При этом необходимо, чтобы в архиве WAL сохранялись все файлы WAL, записанные после предыдущей копии. Если размер этих файлов сравним с общим размером файлов базы данных, ускорение будет менее значительным, но размер копии будет всё же меньше.
- Копирование изменений. В режиме `PTRACK` Postgres Pro отслеживает изменения страниц на лету. Чтобы он работал, не требуется производить непрерывное архивирование WAL. При каждом изменении страницы отношения она помечается в специальной карте `PTRACK`. Это отслеживание привносит небольшие издержки в работу сервера, но значительно ускоряет инкрементальное резервное копирование.

pg_probackup может производить копирование только работающего экземпляра, а для обеспечения целостности таких копий требуется копировать WAL. Поэтому вне зависимости от выбранного режима копирования (FULL, PAGE или DELTA), чтобы pg_probackup мог получить полноценную копию, нужно выбрать один из следующих *режимов доставки WAL*:

- ARCHIVE. В таком режиме целостность копий обеспечивается посредством непрерывного архивирования. Это режим доставки WAL по умолчанию.
- STREAM. Такие резервные копии включают все файлы, необходимые для восстановления целостного состояния кластера на момент создания копии. Вне зависимости от того, осуществляется ли непрерывное архивирование, необходимые для восстановления сегменты WAL считываются по протоколу потоковой репликации во время резервного копирования и включаются в состав резервной копии. Поэтому такие резервные копии называются *автономными* или *самодостаточными*.

Ограничения

В настоящее время pg_probackup имеет следующие ограничения:

- pg_probackup работает с серверами PostgreSQL версии 9.5 или новее.
- Режим удалённого сервера на платформе Windows не поддерживается.
- В Unix-системах копию базы Postgres Pro версии 10 и ниже можно сделать только от имени того пользователя, который запускает сервер Postgres Pro. Например, если сервер Postgres Pro запускает пользователь `postgres`, команду `backup` также должен выполнять пользователь `postgres`. Для удовлетворения этого требования в случае использования удалённого режима и SSH в параметре `--remote-user` необходимо передать `postgres`.
- С Postgres Pro 9.5 функции `pg_create_restore_point(text)` и `pg_switch_xlog()` могут выполняться, только если пользователь резервного копирования имеет права суперпользователя. Поэтому резервное копирование кластера с низкой активностью в WAL обычным пользователем может занять больше времени, чем копирование того же кластера с правами суперпользователя.
- На сервере Postgres Pro, где была сделана копия, и на сервере, где она будет восстанавливаться, должны быть одинаковые значения параметров block_size и wal_block_size и одинаковая основная версия. В зависимости от конфигурации кластера, Postgres Pro может накладывать дополнительные ограничения, например, по архитектуре процессора и версии libc/libicu.

Установка и подготовка

Установив pg_probackup, выполните следующие действия:

- Инициализируйте каталог резервных копий.
- Добавьте копируемый экземпляр в каталог копий.
- Настройте кластер баз данных для использования pg_probackup.
- Настройте SSH для выполнения операций pg_probackup в удалённом режиме, если вы планируете его использовать.

Инициализация каталога резервных копий

pg_probackup сохраняет все файлы копируемых данных и WAL в соответствущих подкаталогах каталога резервных копий.

Для инициализации каталога резервных копий выполните команду:

```
pg_probackup init -B каталог_копий
```

здесь *каталог_копий* — это каталог, предназначенный для резервных копий. Если *каталог_копий* уже существует, он должен быть пустым. В противном случае pg_probackup выдаст ошибку.

Пользователь, запускающий pg_probackup, должен иметь полный доступ к *каталогу_копий*.

pg_probackup создаёт *каталог_копий* со следующими подкаталогами:

- `wal/` — каталог для файлов WAL.
- `backups/` — каталог для файлов резервных копий.

Проинициализировав каталог резервных копий, вы можете добавить определение копируемого экземпляра.

Определение копируемого экземпляра

pg_probackup может сохранять резервные копии разных кластеров баз данных в одном каталоге резервных копий. Для создания необходимых подкаталогов вы должны определить копируемый экземпляр в каталоге копий для каждого кластера баз данных, копию которого вы будете делать.

Для определения копируемого экземпляра выполните команду:

```
pg_probackup add-instance -B каталог_копий -D каталог_данных --instance имя_экземпляра
 [параметры_удалённого_режима]
```

Здесь:

- *каталог_данных* — каталог, содержащий данные кластера, копию которого вы хотите сделать. Для подготовки и использования pg_probackup необходимо иметь право записи в этот каталог.
- *имя_экземпляра* — это имя подкаталогов, в которых будут храниться файлы копируемых данных и WAL для этого кластера.
- параметры_удалённого_режима должны задаваться дополнительно, если *каталог_данных* располагается удалённо.

pg_probackup создаёт подкаталоги *имя_экземпляра* в каталогах `backups/` и `wal/` каталога резервных копий. Каталог `backups/`*имя_экземпляра* содержит файл конфигурации `pg_probackup.conf` с параметрами pg_probackup, относящимися к данному экземпляру копии. Если этой команде передать параметры_удалённого_режима, они будут добавлены в `pg_probackup.conf`.

Более подробно тонкая настройка pg_probackup описывается в Подразделе «Настройка pg_probackup».

Пользователь, запускающий pg_probackup, должен иметь полный доступ к *каталогу_копий* и как минимум доступ на чтение всего содержимого *каталога_данных*. Если вы зададите путь к каталогу копий в переменной окружения `BACKUP_PATH`, соответствующий параметр в командах pg_probackup можно не указывать.

Настройка кластера баз данных

Хотя pg_probackup можно использовать от имени суперпользователя, рекомендуется создать отдельную роль с минимальными правами, необходимыми для выбранной стратегии копирования. В этих инструкциях по настройке такой ролью служит роль `backup`.

Для выполнения резервного копирования роль `backup` должна иметь следующие разрешения на сервере Postgres Pro (только в базе данных, к которой **производится подключение**):

Для Postgres Pro 9.5:

```
BEGIN;
CREATE ROLE backup WITH LOGIN;
GRANT USAGE ON SCHEMA pg_catalog TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.current_setting(text) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.set_config(text, text, boolean) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_is_in_recovery() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_start_backup(text, boolean) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_stop_backup() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_create_restore_point(text) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_switch_xlog() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_current() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_current_snapshot() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_snapshot_xmax(txid_snapshot) TO backup;
COMMIT;
```

Для Postgres Pro 9.6:

```
BEGIN;
CREATE ROLE backup WITH LOGIN;
GRANT USAGE ON SCHEMA pg_catalog TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.current_setting(text) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.set_config(text, text, boolean) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_is_in_recovery() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_start_backup(text, boolean, boolean) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_stop_backup(boolean) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_create_restore_point(text) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_switch_xlog() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_last_xlog_replay_location() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_current() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_current_snapshot() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_snapshot_xmax(txid_snapshot) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_control_checkpoint() TO backup;
COMMIT;
```

Для Postgres Pro версии 10 или новее:

```
BEGIN;
CREATE ROLE backup WITH LOGIN;
GRANT USAGE ON SCHEMA pg_catalog TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.current_setting(text) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.set_config(text, text, boolean) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_is_in_recovery() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_start_backup(text, boolean, boolean) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_stop_backup(boolean, boolean) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_create_restore_point(text) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_switch_wal() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_last_wal_replay_lsn() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_current() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_current_snapshot() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_snapshot_xmax(txid_snapshot) TO backup;
```

```
GRANT EXECUTE ON FUNCTION pg_catalog.pg_control_checkpoint() TO backup;
COMMIT;
```

В файле pg_hba.conf разрешите подключение к кластеру баз данных пользователю с именем `backup`.

Программа pg_probackup должна читать непосредственно файлы кластера, поэтому запускать pg_probackup (или подключаться к нему удалённо) нужно от имени пользователя ОС, который имеет доступ на чтение всех файлов и каталогов внутри каталога данных кластера (`PGDATA`), подлежащего копированию.

В зависимости от того, будете ли вы делать самодостаточные или архивные копии, конфигурация кластера Postgres Pro будет разной (об особенностях рассказывается ниже). Чтобы получить копию кластера баз данных с ведомого сервера, запустить pg_probackup в удалённом режиме или создать резервную копию PTRACK, требуется дополнительная настройка.

Подробнее об этом рассказывается в подразделах Настройка потокового резервного копирования, Настройка непрерывного архивирования WAL, Настройка копирования с резервного сервера, Настройка удалённого режима, Настройка частичного восстановления и Настройка копирования PTRACK.

Настройка потокового резервного копирования

Чтобы настроить кластер для потокового резервного копирования, выполните следующие действия:

- Дайте право `REPLICATION` роли `backup`:

  ```
  ALTER ROLE backup WITH REPLICATION;
  ```

- В файле pg_hba.conf разрешите выполнение репликации для роли `backup`.
- Установите для параметра max_wal_senders достаточно большое значение, предусматривающее минимум одно подключение для процесса резервного копирования.
- Задайте для параметра wal_level значение выше `minimal`.

Если вы намерены выполнять страничное резервное копирование в потоковом режиме или восстановление на момент времени с потоковыми копиями, вам тем не менее надо будет настроить архивирование WAL, как описано в подразделе Настройка непрерывного архивирования WAL.

После этих подготовительных действий вы можете делать резервные копии в режимах FULL, PAGE, DELTA и PTRACK, используя потоковую доставку WAL.

> **Примечание**
>
> Если вы намерены использовать .pgpass для прохождения аутентификации при выполнении копирования в потоковом режиме, файл `.pgpass` должен содержать учётные данные для подключения к базе данных `replication`. Например: `pghost:5432:replication:backup_user:my_strong_password`

Настройка непрерывного архивирования WAL

Для выполнения копирования в режиме PAGE, восстановления на момент времени и создания резервных копий с использованием режима доставки WAL ARCHIVE должно осуществляться непрерывное архивирование WAL. Чтобы настроить непрерывное архивирование, выполните следующие действия:

- Задайте для параметра wal_level значение выше `minimal`.
- Если вы настраиваете резервное копирование на ведущем сервере, параметр archive_mode должен иметь значение `on` или `always`. Для выполнения резервного копирования на ведомом требуется значение `always`.

- Установите параметр archive_command:

```
archive_command = '"путь_инсталляции/pg_probackup" archive-push -B "каталог_копий"
 --instance имя_экземпляра --wal-file-name=%f [параметры_удалённого_режима]'
```

Здесь *путь_инсталляции* — путь к каталогу установленной версии pg_probackup, которую вы хотите использовать, *каталог_копий* и *имя_экземпляра* должны указывать на уже проинициализированный для данного кластера БД копируемый экземпляр, а параметры_удалённого_режима должны задаваться только в случае расположения архива WAL в удалённой системе. Подробнее все возможные параметры `archive-push` рассматриваются в описании archive-push.

После этих подготовительных действий вы сможете использовать режим доставки WAL ARCHIVE и делать резервные копии в режиме PAGE, а также выполнять восстановление на момент времени.

Вы можете просмотреть текущее состояние архива WAL, воспользовавшись командой show. За подробностями обратитесь к Подразделу «Просмотр оглавления архива WAL».

Если вы планируете выполнять страничное копирование и/или делать копии с ведомого сервера, используя режим доставки WAL ARCHIVE, при недостаточной транзакционной активности может потребоваться долго ждать заполнения очередного сегмента WAL. Чтобы ограничить время ожидания, вы можете воспользоваться параметром Postgres Pro archive_timeout на **ведущем** сервере. Значение этого параметра должно быть меньше значения `--archive-timeout` (по умолчанию 5 минут), чтобы заполненный сегмент успел передаться ведомому серверу и попасть в архив WAL, прежде чем копирование прервётся по тайм-ауту, заданному параметром `--archive-timeout`.

> **Примечание**
>
> Вместо использования команды pg_probackup archive-push вы можете воспользоваться любым другим средством, при условии, что в процессе непрерывного архивирования сегменты WAL будут попадать в каталог *каталог_копий*/wal/*имя_экземпляра*. Для сжатия сегментов, если в нём есть потребность, должен использоваться алгоритм `gzip`, а сжатые файлы сегментов должны иметь расширение `.gz`.

> **Примечание**
>
> Организовать непрерывное архивирование можно не только с помощью параметров `archive_mode` и `archive_command`, но и применяя утилиту pg_receivewal. В этом случае аргумент pg_receivewal `-D` *каталог* должен указывать на каталог *каталог_копий*/wal/*имя_экземпляра*. Программа pg_probackup принимает сжатые WAL, которые может сохранять pg_receivewal. Стратегию архивирования «без потерь» можно реализовать только с использованием pg_receivewal.

Настройка копирования с ведомого сервера

С Postgres Pro 9.6 или новее pg_probackup может копировать данные с ведомого сервера. Для этого требуется дополнительная настройка:

- Установите на ведомом сервере для параметра hot_standby значение `on`.
- На ведущем сервере установите для параметра full_page_writes значение `on`.
- Для получения самодостаточных копий с ведомого сервера выполните действия, описанные в подразделе Настройка потокового копирования.
- Для получения архивных копий с ведомого сервера выполните действия, описанные в подразделе Настройка непрерывного архивирования WAL.

После этих подготовительных действий вы можете делать резервные копии с ведомого сервера в режимах FULL, PAGE, DELTA или PTRACK, используя режим доставки WAL ARCHIVE или STREAM.

Копирование с ведомого сервера имеет следующие ограничения:

- Если ведомый сервер повышается до ведущего во время копирования, копирование прерывается с ошибкой.
- Все необходимые для резервной копии записи WAL должны содержать полные страницы. Для этого требуется включить режим `full_page_writes` на ведущем сервере и отказаться от использования в archive_command утилит, удаляющих полные страницы из файлов WAL, как, например, pg_compresslog.

Настройка проверки целостности кластера

Для логической проверки кластера БД требуется дополнительная настройка (предполагается, что настройка производится для роли `backup`):

- Установите расширение amcheck или *amcheck_next* **в каждой базе** кластера:

  ```
  CREATE EXTENSION amcheck;
  ```

- Дайте роли `backup` **в каждой базе данных** кластера следующие права:

```
GRANT SELECT ON TABLE pg_catalog.pg_am TO backup;
GRANT SELECT ON TABLE pg_catalog.pg_class TO backup;
GRANT SELECT ON TABLE pg_catalog.pg_database TO backup;
GRANT SELECT ON TABLE pg_catalog.pg_namespace TO backup;
GRANT SELECT ON TABLE pg_catalog.pg_extension TO backup;
GRANT EXECUTE ON FUNCTION bt_index_check(regclass) TO backup;
GRANT EXECUTE ON FUNCTION bt_index_check(regclass, bool) TO backup;
GRANT EXECUTE ON FUNCTION bt_index_check(regclass, bool, bool) TO backup;
```

Настройка частичного восстановления

Если вы планируете производить частичное восстановление, выполните следующее дополнительное действие:

- Дайте право на чтение `pg_catalog.pg_database` роли `backup` в той базе данных, через которую **выполняется подключение** к серверу Postgres Pro:

  ```
  GRANT SELECT ON TABLE pg_catalog.pg_database TO backup;
  ```

Настройка удалённого режима

Программа pg_probackup поддерживает удалённый режим, в котором резервное копирование, восстановление и архивирование WAL могут выполняться удалённо. В этом режиме каталог резервных копий находится в локальной системе, а целевой экземпляр Postgres Pro работает на удалённом компьютере. В настоящее время это поддерживается с использованием только одного сетевого протокола — SSH.

Настройте SSH

Если вы хотите использовать pg_probackup в удалённом режиме (применяя SSH), выполните следующие действия:

- Установите pg_probackup в обеих системах: `backup_host` (сервер резервного копирования) и `db_host` (сервер баз данных).
- Для организации связи между узлами настройте SSH-соединение без пароля для подключения пользователя `backup` на компьютере `backup_host` к серверу `db_host` под именем `postgres`:

  ```
  [backup@backup_host] ssh-copy-id postgres@db_host
  ```

- Если вы будете использовать непрерывное архивирование WAL, настройте подключение по SSH пользователя `backup` в системе `backup_host` к системе `db_host` под именем пользователя `postgres` без указания пароля:

  ```
  [postgres@db_host] ssh-copy-id backup@backup_host
  ```

Здесь:

- `backup_host` — система, в которой находится *каталог копий*.
- `db_host` — система, в которой функционирует кластер Postgres Pro.
- `backup` — пользователь в системе `backup_host`, от имени которого запускается pg_probackup.
- `postgres` — пользователь в системе `db_host`, от имени которого запускается кластер Postgres Pro.

pg_probackup работает в удалённом режиме по протоколу SSH следующим образом:

- В удалённом режиме поддерживаются только следующие команды: add-instance, backup, restore, catchup, archive-push и archive-get.
- Для работы в удалённом режиме необходимо, чтобы программа pg_probackup была установлена и в локальной, и в удалённой системе, при этом одной и той же версии.
- При запуске в удалённом режиме основной процесс pg_probackup в локальной системе подключается к удалённой по протоколу SSH и запускает в удалённой системе один или несколько агентов, которые называются *удалёнными агентами*. Количество удалённых агентов определяется значением параметра `-j`/`--threads`.
- Основной процесс pg_probackup использует удалённые агенты для передачи данных между локальной и удалённой системой и для обращения к её файлам.
- Удалённые агенты стараются минимизировать сетевой трафик и количество операций взаимодействия узлов.
- Основной процесс обычно запускается в системе *`backup_host`* и подключается к системе *`db_host`*, но в случае команд `archive-push` и `archive-get` основной процесс запускается в системе *`db_host`* и подключается к *`backup_host`*.
- После завершения передачи данных удалённые агенты завершаются и подключения SSH закрываются.
- Если какой-либо удалённый агент столкнулся с ошибкой, все агенты завершаются, а основной процесс pg_probackup выдаёт информацию о возникшей ошибке и также нештатно завершается.
- Сжатие данных всегда осуществляется в системе *`db_host`*, а распаковка — в системе *`backup_host`*.

> **Примечание**
>
> Вы можете задать дополнительные *ограничительные параметры* SSH для защиты целевой системы в случае компрометации используемой учётной записи.

Настройка копирования PTRACK

> **Примечание**
>
> Версии PTRACK ниже 2.0 признаны устаревшими и не поддерживаются. В Postgres Pro Standard и Postgres Pro Enterprise версии 11.9.1 и новее входит обновлённый PTRACK 2.0. Обновите ваш сервер, чтобы избежать проблем с резервными копиями, которые вы будете делать в дальнейшем, и обязательно сделайте копии ваших кластеров с обновлённым PTRACK, так как копии, полученные с использованием PTRACK 1.x, могут быть испорченными.

Если вы намерены использовать режим копирования PTRACK, выполните описанные далее дополнительные действия. Роль, от имени которой будет выполняться копирование PTRACK (в примерах ниже это роль `backup`), должна иметь доступ ко всем базам данных в кластере.

Для Postgres Pro версии 11 или новее:

1. Создайте расширение PTRACK:

```
CREATE EXTENSION ptrack;
```

2. Чтобы включить отслеживание изменений страниц, задайте для параметра `ptrack_map_size` положительное целое значение и перезапустите сервер.

   Для оптимальной производительности рекомендуется задавать `ptrack.map_size` равным *`N / 1024`*, где *`N`* — объём кластера Postgres Pro в мегабайтах. Если он будет иметь меньшее значение, это увеличит вероятность наложения информации разных блоков в карте PTRACK, что повлечёт ложные положительные результаты при определении изменённых блоков и, как следствие, увеличение размера инкрементальной копии, так как в копию будут попадать и фактически неизменённые блоки. Использовать значения `ptrack.map_size`, превышающие 1024, не рекомендуется, хотя PTRACK поддерживает большие карты.

> **Примечание**
>
> В случае изменения значения `ptrack.map_size` ранее созданный файл карты PTRACK очищается, и отслеживание новых блоков начинается с начала. Таким образом, прежде чем создавать новые инкрементальные копии в режиме PTRACK после изменения `ptrack.map_size` необходимо сделать новую полную копию кластера.

Использование

Создание резервной копии

Для создания резервной копии выполните следующую команду:

```
pg_probackup backup -B каталог_копий --instance имя_экземпляра -b режим_копирования
```

Здесь *`режим_копирования`* может быть следующим:

- FULL — создаётся полная резервная копия, содержащая все файлы данных кластера, необходимые для его восстановления.
- DELTA — считываются все файлы данных в каталоге данных и создаётся инкрементальная копия для страниц, изменённых со времени предыдущего копирования.
- PAGE — создаётся инкрементальная копия, содержащая только те страницы, которые фигурируют в файлах WAL, записанных после предыдущей полной или инкрементальной копии. Из файлов данных при этом считываются только изменённые страницы.
- PTRACK — создаётся инкрементальная резервная копия со страницами, изменения в которых отслеживались на лету.

При восстановлении кластера из инкрементальной копии pg_probackup использует родительскую полную копию и все инкрементальные копии между ними, которые в совокупности образуют «цепочку копий». Таким образом, прежде чем делать инкрементальные копии, необходимо сделать как минимум одну полную.

Режим ARCHIVE

Режим ARCHIVE используется в качестве режима доставки WAL по умолчанию.

Например, чтобы сделать полную копию в режиме доставки WAL ARCHIVE, выполните:

```
pg_probackup backup -B каталог_копий --instance имя_экземпляра -b FULL
```

Резервное копирование ARCHIVE требует организации непрерывного архивирования, посредством которого считываются сегменты WAL, требующиеся для восстановления согласованного состояния кластера на момент создания копии.

В процессе резервного копирования pg_probackup помещает файлы WAL, содержащие записи от `Start LSN` до `Stop LSN`, в каталог *`каталог_копий`*`/wal/`*`имя_экземпляра`*. pg_probackup также проверяет, что записи WAL между `Start LSN` и `Stop LSN` корректно читаются. Это позволяет исключить риск хранения в архиве повреждённого WAL.

Режим STREAM

Режим STREAM может использоваться в качестве альтернативного режима доставки WAL.

Например, чтобы сделать полную резервную копию в потоковом режиме, добавьте флаг `--stream` к команде, показанной выше:

```
pg_probackup backup -B каталог_копий --instance имя_экземпляра -b FULL --stream --temp-slot
```

Необязательный параметр `--temp-slot` обеспечивает наличие необходимых сегментов в случае прокрутки WAL до завершения резервного копирования.

В отличие от копий ARCHIVE, копии типа STREAM включают все сегменты WAL, необходимые для восстановления согласованного состояния кластера на момент создания копии.

В процессе выполнения команды backup программа pg_probackup передаёт файлы WAL, содержащие записи от `Start LSN` до `Stop LSN` в каталог *каталог_копий*/backups/*имя_экземпляра*/*ид_резервной_копии*/database/pg_wal. Чтобы исключить риск архивирования испорченных файлов WAL, pg_probackup также проверяет, что записи WAL от `Start LSN` до `Stop LSN` прочитываются корректно.

Даже если вы используете непрерывное архивирование, копирование в режиме STREAM может быть полезно в следующих случаях:

- Копии типа STREAM могут быть восстановлены на сервере, не имеющем файлового доступа к архиву WAL.
- Копии типа STREAM позволяют восстановить состояние кластера на тот момент времени, для которого уже нет файлов WAL.
- Копирование в режиме STREAM можно производить с ведомого сервера, не дожидаясь заполнения сегментов WAL, что могло бы занять длительное время при небольшом объёме трафика WAL.

Проверка страниц

Когда в кластере БД включены контрольные суммы, pg_probackup использует их для проверки целостности файлов данных в процессе резервного копирования. При чтении каждой страницы pg_probackup проверяет, совпадает ли вычисленная сумма с контрольной суммой, хранящейся в заголовке страницы. Это гарантирует, что в кластере Postgres Pro и самой резервной копии не содержатся испорченные страницы. Заметьте, что pg_probackup читает файлы данных непосредственно из файловой системы, поэтому при активной записи в момент копирования возможны ложные выявления некорректных контрольных сумм из-за частичной записи. В случае несовпадения контрольной суммы страница считывается повторно, и контрольная сумма проверяется ещё раз.

Страница признаётся испорченной, если проверка контрольной суммы не проходит более 100 раз. В этом случае резервное копирование прерывается.

Даже если контрольные суммы не включены, pg_probackup всегда проверяет целостность заголовков страниц.

Внешние каталоги

Чтобы заархивировать каталог, размещённый вне каталога данных, воспользуйтесь необязательным параметром `--external-dirs`, в котором можно задать путь к нужному каталогу. Если вы хотите заархивировать несколько внешних каталогов, их пути нужно разделить двоеточиями в Linux или точками с запятой в Windows.

Например, чтобы в системе Linux включить каталоги `/etc/dir1` и `/etc/dir2` в полную копию экземпляра *имя_экземпляра*, которая будет размещаться в *каталоге_копий*, выполните:

```
pg_probackup backup -B каталог_копий --instance имя_экземпляра -b FULL --external-dirs=/etc/dir1:/etc/dir2
```

Например, чтобы включить каталоги `C:\dir1` и `C:\dir2` в полную копию, в Windows нужно выполнить:

```
pg_probackup backup -B каталог_копий --instance имя_экземпляра -b FULL --external-
dirs=C:\dir1;C:\dir2
```

Для каждого внешнего каталога pg_probackup создаёт отдельный подкаталог в каталоге резервной копии и рекурсивно копирует в него всё содержимое внешнего каталога. Так как внешние каталоги, попадающие в разные резервные копии, не обязательно должны быть одинаковыми, при восстановлении кластера из инкрементальной копии будут восстановлены только те каталоги, которые относятся именно к ней. Внешние каталоги, сохранённые в предыдущих копиях, восстановлены не будут.

Чтобы нужные каталоги включались в каждую резервную копию вашего кластера, их список можно сохранить в файле конфигурации `pg_probackup.conf`, воспользовавшись командой set-config с ключом `--external-dirs`.

Проверка целостности кластера

Чтобы убедиться в отсутствии повреждений в кластере Postgres Pro, выполните следующую команду:

```
pg_probackup checkdb [-B каталог_копий [--instance имя_экземпляра]] [-D каталог_данных]
 [параметры_подключения]
```

Эта команда выполняет физическую проверку всех файлов данных в указанном каталоге данных, проводя проверки целостности заголовков страниц, а также проверяя контрольные суммы на уровне блоков, если в кластере включены контрольные суммы. В случае обнаружения испорченной страницы `checkdb` продолжает работу, пока не будут проверены все страницы в кластере.

По умолчанию pg_probackup выполняет подобную проверку страницы автоматически в процессе создания копии. Команда `checkdb` позволяет при желании проводить проверки страниц в кластере, не создавая резервные копии (при этом даже необязательно настраивать копирование кластера в pg_probackup).

Чтобы провести проверку кластера, приложение pg_probackup должно подключиться к нему. Обычно для получения требуемых параметров подключения достаточно указать связанный с данным кластером копируемый экземпляр. Однако если параметры `-B` и `--instance` опускаются, параметры_соединения и *каталог_данных* необходимо задать в командной строке или в переменных окружения.

Важно понимать, что проверка на физическом уровне не позволяет выявить логические несоответствия, потерю или обнуление блоков или целых файлов и другие подобные аномалии. Провести логическую проверку в некотором объёме позволяют расширения amcheck и *amcheck_next*.

Если вы хотите помимо физической проверки проверить также все индексы во всех базах, используя эти расширения, вы можете передать команде checkdb флаг `--amcheck`:

```
pg_probackup checkdb -D каталог_данных --amcheck [параметры_соединения]
```

Проверку на физическом уровне можно отключить, воспользовавшись флагом `--skip-block-validation`. В этом случае задавать *каталог_копий* и *каталог_данных* не нужно, требуется задать только параметры подключения:

```
pg_probackup checkdb --amcheck --skip-block-validation [параметры_соединения]
```

Логическая проверка может быть более доскональной, если запустить её с ключом `--heapallindexed`. В этом случае будет дополнительно проверено, что в индексе действительно представлены все кортежи кучи, которые должны в него попасть, но это создаст дополнительную нагрузку на процессор, память и подсистему ввода/вывода.

Проверка резервных копий

pg_probackup вычисляет контрольные суммы для всех файлов копии в ходе резервного копирования. Процесс проверки контрольных сумм файлов называется *проверкой целостности*

копии. По умолчанию проверка выполняется сразу после создания резервной копии и непосредственно перед восстановлением для выявления возможных повреждений резервных копий.

Если вы хотите пропустить проверку резервной копии, вы можете передать командам backup и restore флаг `--no-validate`.

Чтобы убедиться, что все необходимые файлы резервных копий имеются в наличии и что, используя их, можно восстановить кластер баз данных, вы можете запустить команду validate с теми же параметрами точки восстановления, с которыми вы будете производить восстановление.

Например, чтобы убедиться, что вы можете восстановить кластер баз данных из резервной копии, остановившись на транзакции с идентификатором 4242, выполните команду:

```
pg_probackup validate -B каталог_копий --instance имя_экземпляра --recovery-target-xid=4242
```

Если проверка проходит успешно, pg_probackup выдаёт сообщение об этом. В случае же неудачи вы получите сообщение об ошибке с указанием точного времени, идентификатора транзакции и значения LSN, до которого возможно восстановление.

Если вы укажете *идентификатор копии* в ключе `-i/--backup-id`, будет проверена только резервная копия с указанным идентификатором. Если *идентификатор копии* указывается вместе с параметрами точки восстановления, команда `validate` проверит, возможно ли восстановить указанную резервную копию до заданной точки.

Например, чтобы убедиться, что вы можете восстановить кластер баз данных из резервной копии с идентификатором `PT8XFX` до заданного момента времени, выполните команду:

```
pg_probackup validate -B каталог_копий --instance имя_экземпляра -i PT8XFX --recovery-target-time="2017-05-18 14:18:11+03"
```

Если вы укажете *ид_резервной_копии*, относящийся к инкрементальной копии, будут проверены все нужные ей родительские копии, начиная с полной.

Если вы опустите все параметры, будут проверены все резервные копии.

Восстановление кластера

Чтобы восстановить кластер баз данных из резервной копии, выполните команду restore как минимум со следующими параметрами:

```
pg_probackup restore -B каталог_копий --instance имя_экземпляра -i ид_резервной_копии
```

Здесь:

- *каталог_копий* — каталог, в котором хранятся все файлы резервных копий и метаданные.
- *имя_экземпляра* — имя экземпляра резервной копии кластера, которая будет восстановлена.
- *ид_резервной_копии* определяет, из какой резервной копии будет восстановлен кластер. Если этот параметр опускается, pg_probackup использует последнюю подходящую копию для заданного экземпляра. Если вы выбираете для восстановления инкрементальную копию, pg_probackup автоматически восстанавливает нижележащую полную копию и затем последовательно применяет все необходимые добавления.

После окончания работы команды `restore` запустите службу базы данных.

Если вы восстанавливаете копию ARCHIVE, выполняете восстановление PITR или передаёте флаг `--restore-as-replica` команде `restore` для настройки ведомого сервера, pg_probackup создаёт файл конфигурации восстановления после копирования всех файлов данных в целевой каталог. Этот файл включает необходимые для восстановления параметры, за исключением пароля, заданного в primary_conninfo; если он требуется, его нужно дополнительно задать вручную или воспользоваться параметром `--primary-conninfo`. Для Postgres Pro версии 11 и ниже параметры восстановления сохраняются в файле `recovery.conf`, но для версий Postgres Pro, начиная с 12,

pg_probackup сохраняет эти параметры в файле `probackup_recovery.conf` в каталоге данных и подключает его в `postgresql.auto.conf`.

Если восстанавливалась копия типа STREAM, восстановление завершается сразу, и кластер возвращается в согласованное состояние на момент времени, в который была сделана резервная копия. Для копий типа ARCHIVE Postgres Pro воспроизводит все имеющиеся в архиве сегменты WAL, в результате чего восстанавливается самое последнее состояние кластера на текущей линии времени. Это поведение можно изменить, определив параметры точки восстановления для команды `restore` как описывается в Подразделе «Выполнение восстановления на момент времени (PITR)».

Если кластер, подлежащий восстановлению, содержит табличные пространства, pg_probackup по умолчанию восстанавливает их в исходные расположения. Чтобы сменить расположения табличных пространств, воспользуйтесь параметром `--tablespace-mapping`/`-T`. В противном случае при восстановлении кластера на том же сервере произойдёт ошибка, если эти табличные пространства будут использоваться, так как восстанавливаемые данные нужно будет записать в те же каталоги.

Используя параметр `--tablespace-mapping`/`-T`, вы должны задать абсолютные пути к старому и новому каталогу табличного пространства. Если путь содержит знак равно (=), экранируйте его обратной косой чертой. Данный параметр может указываться неоднократно для перемещения нескольких табличных пространств. Например:

```
pg_probackup restore -B каталог_копий --instance имя_экземпляра -D каталог_данных -j 4
 -i ид_резервной_копии -T каталог_табл_пространства1=новый_каталог_табл_пространства1 -
T каталог_табл_пространства2=новый_каталог_табл_пространства2
```

Восстановление кластера на удалённом компьютере описано в Подразделе «Использование pg_probackup в удалённом режиме».

> **Примечание**
>
> По умолчанию команда restore проверяет указанную резервную копию перед восстановлением кластера. Если вы проводите проверку копий регулярно и хотели бы сэкономить время при восстановлении кластера, вы можете добавить ключ `--no-validate` для пропуска проверки и ускорения восстановления.

Инкрементальное восстановление

Скорость восстановления резервной копии можно значительно увеличить, заменяя в существующем каталоге данных Postgres Pro только некорректные или изменённые страницы. Это можно реализовать, используя параметры инкрементального восстановления с командой restore.

Чтобы восстановить кластер баз данных из резервной копии в инкрементальном режиме, выполните команду restore со следующими параметрами:

```
pg_probackup restore -B каталог_копий --instance имя_экземпляра -D каталог_данных -
I инкрементальный_режим
```

Здесь *инкрементальный_режим* может быть следующим:

- `CHECKSUM` — прочитать все файлы данных в целевом каталоге, проверить заголовок и контрольную сумму каждой страницы и заменить только некорректные страницы, а также страницы, в которых контрольная сумма и LSN отличаются от значений в соответствующей странице в копии. Это самый простой и надёжный инкрементальный режим. Его рекомендуется использовать по умолчанию.
- `LSN` — прочитать файл *pg_control* в каталоге данных, получить из него значения REDO LSN и REDO TLI, позволяющие определить точку в истории (точку сдвига), в которой состояние каталога данных сдвинулись с цепочки резервных копий. Если точка сдвига

находится за пределами истории резервных копий, восстановление прерывается. Если же точка сдвига достижима, прочитываются все файлы данных в каталоге данных, проверяется заголовок и контрольная сумма на каждой странице, а затем заменяются только страницы с неверной контрольной суммой или с LSN, превышающим позицию точки сдвига. В этом режиме обеспечивается увеличенная скорость по сравнению с режимом CHECKSUM, но требуется выполнение двух условий. Во-первых, в целевом каталоге должны быть включены контрольные суммы (см. data_checksums), чтобы не произошло повреждения данных из-за изменения вспомогательных битов. Это условие будет проверяться перед инкрементальным восстановлением — в случае отсутствия контрольных сумм операция прерывается. Во-вторых, необходима синхронность файла `pg_control` с состоянием каталога данных. Это условие нельзя проверить в начале восстановления, так что гарантировать правильность информации в `pg_control` должен пользователь. Поэтому использовать режим LSN рекомендуется не во всех ситуациях, например, он противопоказан, когда файл `pg_control` повреждён или его содержимому нельзя доверять: после выполнения `pg_resetxlog`, после восстановления из копии без процедуры воспроизведения журнала и т. д.
- `NONE` — обычное восстановление без инкрементальных оптимизаций.

Вне зависимости от выбранного инкрементального режима, pg_probackup будет проверять, что с заданным целевым каталогом не работает процесс postmaster и значения `system-identifier` в целевом экземпляре и копии совпадают.

Предположим, вы хотите восстановить старый ведущий сервер в виде реплики после переключения, используя инкрементальное восстановление в режиме LSN:

```
==========================================================================================
 Instance  Version  ID      Recovery Time           Mode   WAL Mode  TLI      Time
 Data    WAL  Zratio  Start LSN    Stop LSN     Status
==========================================================================================
 node       12       QBRNBP  2020-06-11 17:40:58+03  DELTA  ARCHIVE   16/15     40s
 194MB   16MB    8.26  15/2C000028  15/2D000128  OK
 node       12       QBRIDX  2020-06-11 15:51:42+03  PAGE   ARCHIVE   15/15     11s
 18MB   16MB    5.10  14/DC000028  14/DD0000B8  OK
 node       12       QBRIAJ  2020-06-11 15:51:08+03  PAGE   ARCHIVE   15/15     20s
 141MB   96MB    6.22  14/D4BABFE0  14/DA9871D0  OK
 node       12       QBRHT8  2020-06-11 15:45:56+03  FULL   ARCHIVE   15/0   2m:11s
 1371MB  416MB   10.93  14/9D000028  14/B782E9A0  OK

pg_probackup restore -B /backup --instance node -R -I lsn
INFO: Running incremental restore into nonempty directory: "/var/lib/pgsql/12/data"
INFO: Destination directory redo point 15/2E000028 on tli 16 is within reach of backup
 QBRIDX with Stop LSN 14/DD0000B8 on tli 15
INFO: shift LSN: 14/DD0000B8
INFO: Restoring the database from backup at 2020-06-11 17:40:58+03
INFO: Extracting the content of destination directory for incremental restore
INFO: Destination directory content extracted, time elapsed: 1s
INFO: Removing redundant files in destination directory
INFO: Redundant files are removed, time elapsed: 1s
INFO: Start restoring backup files. PGDATA size: 15GB
INFO: Backup files are restored. Transfered bytes: 1693MB, time elapsed: 43s
INFO: Restore incremental ratio (less is better): 11% (1693MB/15GB)
INFO: Restore of backup QBRNBP completed.
```

Примечание

Инкрементальное восстановление возможно только для копий, у которых `program_version` больше или равна 2.4.0.

Частичное восстановление

Если вы подготовились к частичному восстановлению до того, как сделали резервную копию, вы можете восстановить только некоторые базы данных, используя параметры частичного восстановления с командой restore.

Чтобы восстановить только некоторые базы данных, выполните команду `restore` как минимум со следующими параметрами:

```
pg_probackup restore -B каталог_копий --instance имя_экземпляра --db-include=имя_базы
```

Ключ `--db-include` может добавляться многократно. Например, чтобы восстановить только базы `db1` и `db2`, выполните следующую команду:

```
pg_probackup restore -B каталог_копий --instance имя_экземпляра --db-include=db1 --db-include=db2
```

Чтобы исключить одну или несколько баз из числа восстанавливаемых, используйте ключ `--db-exclude`:

```
pg_probackup restore -B каталог_копий --instance имя_экземпляра --db-exclude=имя_базы
```

Ключ `--db-exclude` может добавляться многократно. Например, чтобы при восстановлении исключить базы данных `db1` и `db2`, выполните следующую команду:

```
pg_probackup restore -B каталог_копий --instance имя_экземпляра --db-exclude=db1 --db-exclude=db2
```

Механизм частичного восстановления рассчитывает на то, что в процессе восстановления Postgres Pro наличие пустых файлов не будет проблемой, так как файлы исключённых баз данных восстанавливаются пустыми. После успешного запуска кластера Postgres Pro восстановленные определения исключённых баз данных можно удалить с помощью команды `DROP DATABASE`.

Чтобы максимально быстро разделить один кластер, содержащий несколько баз данных, на разные кластеры, можно выполнить частичное восстановление исходного кластера в виде ведомого, передав ключ `--restore-as-replica` для определённых баз данных.

> **Примечание**
>
> Базы `template0` и `template1` восстанавливаются всегда.

> **Примечание**
>
> Из-за особенностей восстановления, присущих версиям Postgres Pro до 12, при частичном восстановлении кластера Postgres Pro этих версий рекомендуется установить для параметра hot_standby значение `off`. В противном случае при восстановлении возможен сбой.

Выполнение восстановления на момент времени (PITR)

Если вы настраивали непрерывное архивирование WAL до создания резервных копий, вы можете восстановить состояние кластера на любой момент времени (до заданной точки восстановления), используя с командой restore параметры точки восстановления.

Для восстановления на момент времени может использоваться копия типа STREAM или ARCHIVE, но при этом обязательно наличие архива WAL с момента создания копии или более раннего. Если параметр `-i`/`--backup-id` не задан, pg_probackup автоматически выбирает резервную копию, ближайшую к заданной цели восстановления, и начинает процесс восстановления. В противном случае pg_probackup попытается восстановить до заданной цели восстановления именно копию с заданным *ид_резервной_копии*.

- Чтобы восстановить состояние кластера на определённый момент времени, укажите это время в параметре `--recovery-target-time`, в формате timestamp. Например:

  ```
  pg_probackup restore -B каталог_копий --instance имя_экземпляра --recovery-target-time="2017-05-18 14:18:11+03"
  ```

- Чтобы восстановить состояние кластера до определённой транзакции, воспользуйтесь ключом `--recovery-target-xid`:

  ```
  pg_probackup restore -B каталог_копий --instance имя_экземпляра --recovery-target-xid=687
  ```

- Чтобы восстановить состояние кластера до определённой позиции в журнале (LSN), воспользуйтесь ключом `--recovery-target-lsn`:

  ```
  pg_probackup restore -B каталог_копий --instance имя_экземпляра --recovery-target-lsn=16/B374D848
  ```

- Чтобы восстановить состояние кластера до заданной именованной точки восстановления, воспользуйтесь ключом `--recovery-target-name`:

  ```
  pg_probackup restore -B каталог_копий --instance имя_экземпляра --recovery-target-name="before_app_upgrade"
  ```

- Чтобы восстановить последнее возможное состояние, исходя из содержимого архива WAL, передайте в параметре `--recovery-target` значение `latest`:

  ```
  pg_probackup restore -B каталог_копий --instance имя_экземпляра --recovery-target="latest"
  ```

- Чтобы восстановить самое раннее из возможных согласованное состояние кластера, передайте в параметре `--recovery-target` значение `immediate`:

  ```
  pg_probackup restore -B каталог_копий --instance имя_экземпляра --recovery-target='immediate'
  ```

Использование pg_probackup в удалённом режиме

Программа pg_probackup поддерживает работу в удалённом режиме, то есть может выполнять операции `backup` и `restore` удалённо, используя SSH. В этом режиме каталог резервных копий располагается в локальной системе, а целевой экземпляр Postgres Pro работает в удалённой. При этом pg_probackup должен быть установлен в обеих системах.

> **Примечание**
>
> pg_probackup рассчитывает на то, что узлы будут взаимодействовать между собой с использованием SSH без пароля.

Типичная схема его использования выглядит так:

- В системе резервного копирования настройте pg_probackup, как описывается в подразделе Установка и настройка. Для команд add-instance и set-config необходимо задать параметры удалённого сервера, указывающие на сервер с экземпляром Postgres Pro.
- Если вы хотите в удалённом режиме выполнять страничное копирование (PAGE), использовать доставку WAL в режиме ARCHIVE или осуществлять восстановление PITR, настройте непрерывное архивирование WAL с сервера БД в систему резервного копирования, как описано в подразделе Настройка непрерывного архивирования WAL. Для этого в командах archive-push и archive-get требуется задать параметры удалённого сервера, указывающие на сервер, где находится каталог резервных копий.
- Запустите команду backup или restore с параметрами удалённого сервера **в системе резервного копирования**. pg_probackup подключится к удалённой системе по SSH и

соответственно создаст резервную копию в локальной системе или восстановит в удалённой системе ранее сделанную копию.

Например, чтобы создать полную архивную копию кластера Postgres Pro, работающего в удалённой системе с адресом `192.168.0.2`, подключившись к серверу по SSH через порт `2302` с именем пользователя `postgres`, выполните:

```
pg_probackup backup -B каталог_копий --instance имя_экземпляра -b FULL --remote-
user=postgres --remote-host=192.168.0.2 --remote-port=2302
```

Чтобы восстановить последнюю имеющуюся копию в удалённой системе с адресом `192.168.0.2`, подключившись к серверу по SSH через порт `2302` с именем пользователя `postgres`, выполните:

```
pg_probackup restore -B каталог_копий --instance имя_экземпляра --remote-user=postgres
 --remote-host=192.168.0.2 --remote-port=2302
```

Для восстановления архивных копий или восстановления на момент времени в удалённом режиме требуется дополнительная информация: целевой адрес, порт и имя пользователя для установления SSH-подключения **с** сервера БД **к** системе, где находится каталог копий. Эту информацию будет использовать команда `restore_command` для копирования сегментов WAL из архива в каталог Postgres Pro `pg_wal`.

Для передачи этой информации вы можете задать параметры удалённого архива WAL.

Например, для восстановления последней копии в удалённой системе, настроенной для приёма подключения пользователя `postgres` по адресу `192.168.0.2` через порт `2302`, с использованием каталога резервных копий в системе, настроенной для подключения пользователя `backup` по адресу `192.168.0.3` через порт `2303`, выполните:

```
pg_probackup restore -B каталог_копий --instance имя_экземпляра --remote-user=postgres
 --remote-host=192.168.0.2 --remote-port=2302 --archive-host=192.168.0.3 --archive-
port=2303 --archive-user=backup
```

Переданные аргументы будут использоваться при составлении команды `restore_command`:

```
restore_command = '"путь_инсталляции/pg_probackup" archive-get -B "каталог_копий" --
instance имя_экземпляра --wal-file-path=%p --wal-file-name=%f --remote-host=192.168.0.3
 --remote-port=2303 --remote-user=backup'
```

Также вы можете воспользоваться ключом `--restore-command` и передать всю команду *restore_command* целиком:

```
pg_probackup restore -B каталог_копий --instance имя_экземпляра --remote-user=postgres
 --remote-host=192.168.0.2 --remote-port=2302 --restore-command='"путь_инсталляции/
pg_probackup" archive-get -B "каталог_копий" --instance имя_экземпляра --wal-file-
path=%p --wal-file-name=%f --remote-host=192.168.0.3 --remote-port=2303 --remote-
user=backup'
```

> **Примечание**
>
> Использование удалённого режима на платформе Windows в настоящее время не поддерживается.

Запуск pg_probackup в параллельных потоках

Команды backup, restore, merge, delete, checkdb и validate могут выполняться в несколько параллельных потоков. Это может существенно ускорять работу pg_probackup при наличии достаточных ресурсов (ядер процессора, производительности дисковой подсистемы и сети).

Параллельным выполнением управляет ключ командной строки `-j/--threads`. Например, чтобы запустить резервное копирование в четыре параллельных потока, выполните:

```
pg_probackup backup -B каталог_копий --instance имя_экземпляра -b FULL -j 4
```

> **Примечание**
>
> Восстановление происходит в параллельном режиме только на этапе копирования данных из каталога копий в каталог данных кластера. При запуске сервера Postgres Pro он должен будет воспроизвести записи из WAL, а это может происходить только последовательно.

Настройка pg_probackup

Проинициализировав каталог резервных копий и добавив определение копируемого экземпляра, вы можете использовать файл `pg_probackup.conf` в каталоге *каталог_копий*`/backups/`*имя_экземпляра* для тонкой настройки конфигурации pg_probackup.

Например, команды backup и checkdb работают через обычное подключение Postgres Pro. Чтобы каждый раз не задавать параметры подключения в командной строке, вы можете определить их в файле конфигурации `pg_probackup.conf` с помощью команды set-config.

> **Примечание**
>
> Редактировать `pg_probackup.conf` вручную **не рекомендуется**.

Изначально `pg_probackup.conf` содержит следующие параметры:

- `PGDATA` — путь к каталогу данных кластера, который будет копироваться.
- `system-identifier` — уникальный идентификатор экземпляра Postgres Pro.

Вы можете дополнительно установить параметры удалённого режима, хранения копий, ведения журнала и сжатия, используя команду `set-config`:

```
pg_probackup set-config -B каталог_копий --instance имя_экземпляра
[--external-dirs=путь_внешнего_каталога] [параметры_удалённого_режима]
 [параметры_соединения] [параметры_хранения] [параметры_журнала]
```

Чтобы просмотреть текущие параметры, выполните эту команду:

```
pg_probackup show-config -B каталог_копий --instance имя_экземпляра
```

Параметры, заданные в `pg_probackup.conf`, можно переопределить, задавая соответствующие переменные окружения или параметры командной строки при вызове команд pg_probackup.

Указание параметров подключения

Если вы определите свойства соединения в файле конфигурации `pg_probackup.conf`, вы можете не указывать параметры соединения во всех последующих командах pg_probackup. Однако если установлены соответствующие переменные окружения, они имеют больший приоритет. Параметры, заданные в командной строке, переопределяют как переменные окружения, так и свойства в файле конфигурации.

Если не задано ничего, используются значения по умолчанию. В частности, pg_probackup пытается подключиться к Unix-сокету локального сервера (или к `localhost` в Windows), а в качестве имени базы данных и имени пользователя выбирает значение переменной окружения `PGUSER` либо имя текущего пользователя ОС.

Управление каталогом резервных копий

С помощью pg_probackup вы можете управлять резервными копиями в командной строке:

- Просматривать имеющиеся резервные копии

- Просматривать оглавление архива WAL
- Проверять резервные копии
- Объединять резервные копии
- Удалять резервные копии

Просмотр информации о резервных копиях

Чтобы просмотреть список существующих копий для каждого экземпляра, выполните команду:

```
pg_probackup show -B каталог_копий
```

pg_probackup выводит список всех имеющихся резервных копий. Например:

```
BACKUP INSTANCE 'node'
============================================================================================
 Instance  Version  ID      Recovery time           Mode    WAL Mode  TLI  Time    Data
   WAL  Zratio  Start LSN   Stop LSN    Status
============================================================================================
 node       10       PYSUE8  2019-10-03 15:51:48+03  FULL    ARCHIVE   1/0   16s  9047kB
  16MB    4.31  0/12000028  0/12000160  OK
 node       10       P7XDQV  2018-04-29 05:32:59+03  DELTA   STREAM    1/1   11s    19MB
  16MB    1.00  0/15000060  0/15000198  OK
 node       10       P7XDJA  2018-04-29 05:28:36+03  PTRACK  STREAM    1/1   21s    32MB
  32MB    1.00  0/13000028  0/13000198  OK
 node       10       P7XDHU  2018-04-29 05:27:59+03  PAGE    STREAM    1/1   15s    33MB
  16MB    1.00  0/11000028  0/110001D0  OK
 node       10       P7XDHB  2018-04-29 05:27:15+03  FULL    STREAM    1/0   11s    39MB
  16MB    1.00  0/F000028   0/F000198   OK
```

Для каждой копии выдаются следующие сведения:

- `Instance` — имя экземпляра.
- `Version` — базовая версия Postgres Pro.
- `ID` — идентификатор резервной копии.
- `Recovery time` — самое ранее время, на которое можно восстановить кластер из данной копии.
- `Mode` — режим, в котором была сделана копия. Возможные значения: `FULL` (полная), `PAGE` (страничная), `DELTA` (инкрементальная), `PTRACK` (копирование изменений).
- `WAL Mode` — режим доставки WAL. Возможные значения: `STREAM` (потоковый) и `ARCHIVE` (архивный).
- `TLI` — идентификаторы линии времени текущей копии и её родителя.
- `Time` — время, за которое была выполнена данная копия.
- `Data` — объём файлов данных в этой копии. Это значение не включает в себя объём файлов WAL. Для копий, сделанных в режиме STREAM, общий размер можно рассчитать, сложив значения `Data` и `WAL`.
- `WAL` — размер несжатых файлов WAL, которые должны быть применены в процессе восстановления копии для достижения согласованного состояния.
- `Zratio` — коэффициент сжатия, вычисленный как отношение «uncompressed-bytes» (объём несжатых данных в байтах) к «data-bytes» (итоговый объём данных).
- `Start LSN` — последовательный номер в журнале WAL, соответствующий началу процесса копирования. С этой позиции накатываются изменения (REDO) в процессе восстановления Postgres Pro.
- `Stop LSN` — последовательный номер в журнале WAL, соответствующий окончанию процесса копирования. Это позиция точки согласованности при восстановлении Postgres Pro.

- `Status` — состояние резервной копии. Возможные варианты:
  - `OK` — резервная копия сделана и пригодна к использованию.
  - `DONE` — резервная копия сделана, но не проверена.
  - `RUNNING` — резервное копирование выполняется.
  - `MERGING` — резервная копия объединяется.
  - `MERGED` — файлы резервной копии были успешно обработаны в процессе объединения копий, но её метаданные ещё изменяются. Это состояние могут иметь только полные резервные копии.
  - `DELETING` — файлы резервной копии удаляются.
  - `CORRUPT` — некоторые файлы резервной копии повреждены.
  - `ERROR` — резервное копирование было прервано из-за неожиданной ошибки.
  - `ORPHAN` — резервная копия непригодна к использованию, так как её родительская копия испорчена или отсутствует.

Восстановить кластер из копии можно только для копий с состоянием `OK` или `DONE`.

Чтобы получить более подробную информацию о копии, укажите в команде `show` её идентификатор:

```
pg_probackup show -B каталог_копий --instance имя_экземпляра -i ид_резервной_копии
```

Пример вывода:

```
#Configuration
backup-mode = FULL
stream = false
compress-alg = zlib
compress-level = 1
from-replica = false

#Compatibility
block-size = 8192
wal-block-size = 8192
checksum-version = 1
program-version = 2.1.3
server-version = 10

#Result backup info
timelineid = 1
start-lsn = 0/04000028
stop-lsn = 0/040000f8
start-time = '2017-05-16 12:57:29'
end-time = '2017-05-16 12:57:31'
recovery-xid = 597
recovery-time = '2017-05-16 12:57:31'
expire-time = '2020-05-16 12:57:31'
data-bytes = 22288792
wal-bytes = 16777216
uncompressed-bytes = 39961833
pgdata-bytes = 39859393
status = OK
parent-backup-id = 'PT8XFX'
primary_conninfo = 'user=backup passfile=/var/lib/pgsql/.pgpass port=5432
 sslmode=disable sslcompression=1 target_session_attrs=any'
```

В расширенном выводе представлены дополнительные атрибуты:

- `compress-alg` — алгоритм сжатия, используемый при получении резервной копии. Возможные значения: `zlib`, `pglz` и `none` (сжатие не производилось).

- `compress-level` — уровень сжатия, применяемый в процессе резервного копирования.
- `from-replica` — признак того, что копия была сделана с ведомого сервера. Возможные значения: `1`, `0`.
- `block-size` — значение параметра block_size, установленное в кластере Postgres Pro, с которого была сделана копия.
- `checksum-version` — признак включения параметра data_checksums в исходном кластере Postgres Pro. Возможные значения: `1`, `0`.
- `program-version` — полная версия программы pg_probackup, которая создала эту копию.
- `start-time` — время начала резервного копирования.
- `end-time` — время окончания резервного копирования.
- `expire-time` — время, когда закреплённая копия может быть ликвидирована в соответствии с политикой хранения. Этот атрибут имеется только у закреплённых копий.
- `uncompressed-bytes` — размер файлов данных до добавления заголовков страниц и сжатия. В случае использования сжатия оценить его эффективность можно, сопоставив объём `uncompressed-bytes` (байтов несжатых данных) с `data-bytes` (байтов данных).
- `pgdata-bytes` — размер файлов данных Postgres Pro на момент копирования. Эффективность инкрементального метода копирования можно оценить, сопоставив объём `pgdata-bytes` (байтов в PGDATA) с `uncompressed-bytes` (байтов несжатых данных).
- `recovery-xid` — идентификатор транзакции, соответствующей моменту окончания резервного копирования.
- `parent-backup-id` — идентификатор родительской копии. Определён только для инкрементальных копий.
- `primary_conninfo` — параметры подключения libpq, с использованием которых производилось подключение к кластеру Postgres Pro для получения этой резервной копии. Пароль в эти параметры не включается.
- `note` — текстовое примечание, связанное с копией.
- `content-crc` — контрольная сумма файла `backup_content.control`, рассчитанная по алгоритму CRC32. Она позволяет выявить повреждение метаданных копии.

Вы также можете получить подробную информацию о резервной копии в формате JSON:

```
pg_probackup show -B каталог_копий --instance имя_экземпляра --format=json -i backup_id
```

Пример вывода:

```
[
  {
      "instance": "node",
      "backups": [
          {
              "id": "PT91HZ",
              "parent-backup-id": "PT8XFX",
              "backup-mode": "DELTA",
              "wal": "ARCHIVE",
              "compress-alg": "zlib",
              "compress-level": 1,
              "from-replica": false,
              "block-size": 8192,
              "xlog-block-size": 8192,
              "checksum-version": 1,
              "program-version": "2.1.3",
              "server-version": "10",
              "current-tli": 16,
              "parent-tli": 2,
              "start-lsn": "0/8000028",
              "stop-lsn": "0/8000160",
              "start-time": "2019-06-17 18:25:11+03",
              "end-time": "2019-06-17 18:25:16+03",
```

```
                "recovery-xid": 0,
                "recovery-time": "2019-06-17 18:25:15+03",
                "data-bytes": 106733,
                "wal-bytes": 16777216,
                "primary_conninfo": "user=backup passfile=/var/lib/pgsql/.pgpass
 port=5432 sslmode=disable sslcompression=1 target_session_attrs=any",
                "status": "OK"
            }
        ]
    }
]
```

Просмотр оглавления архива WAL

Чтобы получить информацию об архиве WAL для каждого экземпляра, выполните команду:

```
pg_probackup show -B каталог_копий [--instance имя_экземпляра] --archive
```

pg_probackup выводит список всех имеющихся файлов WAL, сгруппированных по линиям времени. Например:

```
ARCHIVE INSTANCE 'node'
================================================================================================================================
 TLI  Parent TLI  Switchpoint  Min Segno                 Max Segno                 N
 segments  Size    Zratio  N backups  Status
================================================================================================================================
 5    1           0/B000000    000000050000000000000000B  000000050000000000000000C  2
       685kB   48.00   0          OK
 4    3           0/18000000   000000040000000000000018  00000004000000000000001A  3
       648kB   77.00   0          OK
 3    2           0/15000000   000000030000000000000015  000000030000000000000017  3
       648kB   77.00   0          OK
 2    1           0/B000108    000000020000000000000000B  000000020000000000000015  5
       892kB   94.00   1          DEGRADED
 1    0           0/0          000000010000000000000001  00000001000000000000000A  10
       8774kB  19.00   1          OK
```

Для каждой линии времени выдаются следующие сведения:

- `TLI` — идентификатор линии времени.
- `Parent TLI` — идентификатор линии времени, от которой была ответвлена данная.
- `Switchpoint` — LSN момента, когда эта линия времени ответвилась от родительской.
- `Min Segno` — первый сегмент WAL, относящийся к этой линии времени.
- `Max Segno` — последний сегмент WAL, относящийся к этой линии времени.
- `N segments` — количество сегментов WAL, относящихся к этой линии времени.
- `Size` — объём, который занимают файлы на диске.
- `Zratio` — коэффициент сжатия, вычисляемый по формуле `N segments` * *wal_segment_size* * *wal_block_size* / `Size`.
- `N backups` — число копий, относящихся к этой линии времени. Для получения подробных сведений об этих копиях воспользуйтесь форматом JSON.
- `Status` — состояние архива WAL для этой линии времени. Возможные значения:
  - `OK` — в архиве присутствуют все сегменты WAL между `Min Segno` и `Max Segno`.
  - `DEGRADED` — некоторые сегменты WAL в интервале между `Min Segno` и `Max Segno` отсутствуют. Понять, какие именно, можно, получив отчёт в формате JSON.

Чтобы получить более подробную информацию об архиве WAL в формате JSON, выполните команду:

```
pg_probackup show -B каталог_копий [--instance имя_экземпляра] --archive --format=json
```

Пример вывода:


```
[
  {
      "instance": "replica",
      "timelines": [
          {
              "tli": 5,
              "parent-tli": 1,
              "switchpoint": "0/B000000",
              "min-segno": "00000005000000000000000B",
              "max-segno": "00000005000000000000000C",
              "n-segments": 2,
              "size": 685320,
              "zratio": 48.00,
              "closest-backup-id": "PXS92O",
              "status": "OK",
              "lost-segments": [],
              "backups": []
          },
          {
              "tli": 4,
              "parent-tli": 3,
              "switchpoint": "0/18000000",
              "min-segno": "000000040000000000000018",
              "max-segno": "00000004000000000000001A",
              "n-segments": 3,
              "size": 648625,
              "zratio": 77.00,
              "closest-backup-id": "PXS9CE",
              "status": "OK",
              "lost-segments": [],
              "backups": []
          },
          {
              "tli": 3,
              "parent-tli": 2,
              "switchpoint": "0/15000000",
              "min-segno": "000000030000000000000015",
              "max-segno": "000000030000000000000017",
              "n-segments": 3,
              "size": 648911,
              "zratio": 77.00,
              "closest-backup-id": "PXS9CE",
              "status": "OK",
              "lost-segments": [],
              "backups": []
          },
          {
              "tli": 2,
              "parent-tli": 1,
              "switchpoint": "0/B000108",
              "min-segno": "00000002000000000000000B",
              "max-segno": "000000020000000000000015",
              "n-segments": 5,
              "size": 892173,
              "zratio": 94.00,
              "closest-backup-id": "PXS92O",
```

```
            "status": "DEGRADED",
            "lost-segments": [
                {
                    "begin-segno": "0000000200000000000000D",
                    "end-segno": "0000000200000000000000E"
                },
                {
                    "begin-segno": "00000002000000000000010",
                    "end-segno": "00000002000000000000012"
                }
            ],
            "backups": [
                {
                    "id": "PXS9CE",
                    "backup-mode": "FULL",
                    "wal": "ARCHIVE",
                    "compress-alg": "none",
                    "compress-level": 1,
                    "from-replica": "false",
                    "block-size": 8192,
                    "xlog-block-size": 8192,
                    "checksum-version": 1,
                    "program-version": "2.1.5",
                    "server-version": "10",
                    "current-tli": 2,
                    "parent-tli": 0,
                    "start-lsn": "0/C000028",
                    "stop-lsn": "0/C000160",
                    "start-time": "2019-09-13 21:43:26+03",
                    "end-time": "2019-09-13 21:43:30+03",
                    "recovery-xid": 0,
                    "recovery-time": "2019-09-13 21:43:29+03",
                    "data-bytes": 104674852,
                    "wal-bytes": 16777216,
                    "primary_conninfo": "user=backup passfile=/var/lib/pgsql/.pgpass
port=5432 sslmode=disable sslcompression=1 target_session_attrs=any",
                    "status": "OK"
                }
            ]
        },
        {
            "tli": 1,
            "parent-tli": 0,
            "switchpoint": "0/0",
            "min-segno": "000000010000000000000001",
            "max-segno": "00000001000000000000000A",
            "n-segments": 10,
            "size": 8774805,
            "zratio": 19.00,
            "closest-backup-id": "",
            "status": "OK",
            "lost-segments": [],
            "backups": [
                {
                    "id": "PXS92O",
                    "backup-mode": "FULL",
                    "wal": "ARCHIVE",
                    "compress-alg": "none",
```

```
                    "compress-level": 1,
                    "from-replica": "true",
                    "block-size": 8192,
                    "xlog-block-size": 8192,
                    "checksum-version": 1,
                    "program-version": "2.1.5",
                    "server-version": "10",
                    "current-tli": 1,
                    "parent-tli": 0,
                    "start-lsn": "0/4000028",
                    "stop-lsn": "0/6000028",
                    "start-time": "2019-09-13 21:37:36+03",
                    "end-time": "2019-09-13 21:38:45+03",
                    "recovery-xid": 0,
                    "recovery-time": "2019-09-13 21:37:30+03",
                    "data-bytes": 25987319,
                    "wal-bytes": 50331648,
                    "primary_conninfo": "user=backup passfile=/var/lib/pgsql/.pgpass
port=5432 sslmode=disable sslcompression=1 target_session_attrs=any",
                    "status": "OK"
                }
            ]
        }
    ]
 },
 {
    "instance": "master",
    "timelines": [
        {
            "tli": 1,
            "parent-tli": 0,
            "switchpoint": "0/0",
            "min-segno": "000000010000000000000001",
            "max-segno": "00000001000000000000000B",
            "n-segments": 11,
            "size": 8860892,
            "zratio": 20.00,
            "status": "OK",
            "lost-segments": [],
            "backups": [
                {
                    "id": "PXS92H",
                    "parent-backup-id": "PXS92C",
                    "backup-mode": "PAGE",
                    "wal": "ARCHIVE",
                    "compress-alg": "none",
                    "compress-level": 1,
                    "from-replica": "false",
                    "block-size": 8192,
                    "xlog-block-size": 8192,
                    "checksum-version": 1,
                    "program-version": "2.1.5",
                    "server-version": "10",
                    "current-tli": 1,
                    "parent-tli": 1,
                    "start-lsn": "0/4000028",
                    "stop-lsn": "0/50000B8",
                    "start-time": "2019-09-13 21:37:29+03",
```

```
                    "end-time": "2019-09-13 21:37:31+03",
                    "recovery-xid": 0,
                    "recovery-time": "2019-09-13 21:37:30+03",
                    "data-bytes": 1328461,
                    "wal-bytes": 33554432,
                    "primary_conninfo": "user=backup passfile=/var/lib/pgsql/.pgpass
 port=5432 sslmode=disable sslcompression=1 target_session_attrs=any",
                    "status": "OK"
                },
                {
                    "id": "PXS92C",
                    "backup-mode": "FULL",
                    "wal": "ARCHIVE",
                    "compress-alg": "none",
                    "compress-level": 1,
                    "from-replica": "false",
                    "block-size": 8192,
                    "xlog-block-size": 8192,
                    "checksum-version": 1,
                    "program-version": "2.1.5",
                    "server-version": "10",
                    "current-tli": 1,
                    "parent-tli": 0,
                    "start-lsn": "0/2000028",
                    "stop-lsn": "0/2000160",
                    "start-time": "2019-09-13 21:37:24+03",
                    "end-time": "2019-09-13 21:37:29+03",
                    "recovery-xid": 0,
                    "recovery-time": "2019-09-13 21:37:28+03",
                    "data-bytes": 24871902,
                    "wal-bytes": 16777216,
                    "primary_conninfo": "user=backup passfile=/var/lib/pgsql/.pgpass
 port=5432 sslmode=disable sslcompression=1 target_session_attrs=any",
                    "status": "OK"
                }
            ]
        }
    ]
  }
]
```


В основном в этом формате представлены те же поля, что и в текстовом формате, с некоторыми исключениями:

- Размер выражается в байтах.
- Атрибут `closest-backup-id` содержит идентификатор самой последней доступной копии, принадлежащей к одной из предыдущих линий времени. Вы можете использовать эту копию для восстановления на момент, относящийся к этой линии времени. Если такой копии не существует, данный атрибут будет пустым.
- В массиве `lost-segments` представлены интервалы отсутствующих сегментов на линиях времени в непригодном состоянии (`DEGRADED`). На линиях времени в целостном состоянии `OK` массив `lost-segments` пуст.
- В массиве `backups` перечисляются все резервные копии, относящиеся к данной линии времени. Если к линии времени не относятся резервные копии, этот массив пуст.

Настройка политики хранения

Используя pg_probackup, вы можете реализовать политики хранения резервных копий, в соответствии с которыми могут удаляться лишние копии, очищаться ненужные сегменты WAL.

Кроме того, можно закреплять определённые копии, чтобы они сохранялись независимо от политики, как описано ниже. Эти подходы можно комбинировать произвольным образом.

Удаление ненужных копий

По умолчанию все резервные копии, которые создаёт pg_probackup, сохраняются в предназначенном для них каталоге. Для экономии дискового пространства вы можете настроить политику сохранения копий, чтобы ненужные копии удалялись.

Чтобы настроить политику хранения, задайте одну или несколько следующих переменных в файле `pg_probackup.conf` с помощью команды set-config:

`--retention-redundancy=`*`избыточность`*

Определяет **количество полных резервных копий**, которое должно сохраняться в каталоге копий.

`--retention-window=`*`окно`*

Определяет самый ранний момент времени, на который pg_probackup может выполнить восстановление. В этом параметре задаётся **количество дней** от текущего момента. Например, если `retention-window=7`, должна сохраниться минимум одна копия старее 7 дней, вместе с соответствующими файлами WAL и всеми последующими копиями.

Если установлены параметры `--retention-redundancy` и `--retention-window`, при очистке каталога от ненужных копий принимаются во внимание оба заданных ими условия. Например, если задать параметры `--retention-redundancy=2` и `--retention-window=7`, программа pg_probackup должна сохранить две полные резервные копии, а также все копии, необходимые для восстановления данных, за последние семь дней:

```
pg_probackup set-config -B каталог_копий --instance имя_экземпляра --retention-
redundancy=2 --retention-window=7
```

Чтобы очистить каталог копий в соответствии с политикой хранения, нужно запустить команду delete с флагами сохранения, как показано ниже, или выполнить с этими флагами команду backup. В последнем случае ненужные копии будут удалены сразу после создания новой копии.

Например, чтобы удалить все резервные копии, считающиеся ненужными согласно установленной политике хранения, нужно выполнить следующую команду с флагом `--delete-expired`:

```
pg_probackup delete -B каталог_копий --instance имя_экземпляра --delete-expired
```

Если вы хотите также удалить файлы WAL, которые больше не требуются ни для каких копий, укажите дополнительно `--delete-wal`:

```
pg_probackup delete -B каталог_копий --instance имя_экземпляра --delete-expired --
delete-wal
```

Вы также можете установить или переопределить текущую политику хранения, добавив параметры `--retention-redundancy` и `--retention-window` непосредственно при выполнении команд `delete` или `backup`:

```
pg_probackup delete -B каталог_копий --instance имя_экземпляра --delete-expired --
retention-window=7 --retention-redundancy=2
```

Так как для инкрементальных копий требуется наличие всех родительских полных копий и всех предыдущих инкрементальных копий, даже по истечении срока их хранения эти копии нельзя удалить, пока минимум одна инкрементальная копия в цепочке удовлетворяет политике хранения. Чтобы не хранить устаревшие копии, которые всё ещё нужны для восстановления нужной инкрементальной копии, вы можете объединить их с ней, воспользовавшись ключом `--merge-expired` команды backup или delete.

Предположим, что вы заархивировали экземпляр *`node`* в *`каталог_копий`* со значением параметра `--retention-window`, равным `7`, и на 10 апреля 2019 г. у вас есть следующие копии:

```
BACKUP INSTANCE 'node'
================================================================================================
 Instance  Version  ID      Recovery time           Mode   WAL     TLI  Time    Data
 WAL  Zratio  Start LSN   Stop LSN    Status
================================================================================================
 node       10       P7XDHR  2019-04-10 05:27:15+03  FULL   STREAM  1/0   11s   200MB
 16MB     1.0  0/18000059  0/18000197  OK
 node       10       P7XDQV  2019-04-08 05:32:59+03  PAGE   STREAM  1/0   11s    19MB
 16MB     1.0  0/15000060  0/15000198  OK
 node       10       P7XDJA  2019-04-03 05:28:36+03  DELTA  STREAM  1/0   21s    32MB
 16MB     1.0  0/13000028  0/13000198  OK
 ----------------------------------------------------retention
 window------------------------------------------------------
 node       10       P7XDHU  2019-04-02 05:27:59+03  PAGE   STREAM  1/0   31s    33MB
 16MB     1.0  0/11000028  0/110001D0  OK
 node       10       P7XDHB  2019-04-01 05:27:15+03  FULL   STREAM  1/0   11s   200MB
 16MB     1.0  0/F000028   0/F000198   OK
 node       10       P7XDFT  2019-03-29 05:26:25+03  FULL   STREAM  1/0   11s   200MB
 16MB     1.0  0/D000028   0/D000198   OK
```

Несмотря на то, что копии `P7XDHB` и `P7XDHU` выходят за рамки окна хранения, их нельзя удалить, так как от них зависят последующие инкрементальные копии `P7XDJA` и `P7XDQV`, которые всё ещё нужны. Поэтому, если вы выполните команду `delete` с ключом `--delete-expired`, будет удалена только полная копия `P7XDFT`.

С ключом `--merge-expired` резервная копия `P7XDJA` объединяется с нижележащими `P7XDHU` и `P7XDHB` и становится полной, поэтому хранить две предыдущие просроченные копии больше не нужно:

```
pg_probackup delete -B каталог_копий --instance node --delete-expired --merge-expired
pg_probackup show -B каталог_копий
```

```
BACKUP INSTANCE 'node'
================================================================================================
 Instance  Version  ID      Recovery time           Mode  WAL     TLI  Time    Data
 WAL  Zratio  Start LSN   Stop LSN    Status
================================================================================================
 node       10       P7XDHR  2019-04-10 05:27:15+03  FULL  STREAM  1/0   11s   200MB
 16MB     1.0  0/18000059  0/18000197  OK
 node       10       P7XDQV  2019-04-08 05:32:59+03  PAGE  STREAM  1/0   11s    19MB
 16MB     1.0  0/15000060  0/15000198  OK
 node       10       P7XDJA  2019-04-03 05:28:36+03  FULL  STREAM  1/0   21s    32MB
 16MB     1.0  0/13000028  0/13000198  OK
```

Поле `Time` (Время) для объединённой копии показывает время, которое заняла процедура объединения.

Закрепление резервных копий

Если вам нужно хранить отдельные копии дольше, чем допускает установленная политика хранения, вы можете дополнительно закрепить их на определённое время. Например:

```
pg_probackup set-backup -B каталог_копий --instance имя_экземпляра -
i ид_резервной_копии --ttl=30d
```

Эта команда задаёт срок хранения заданной резервной копии 30 дней, начиная с момента, указанного в атрибуте `recovery-time` этой копии.

Вы также можете явно задать время истечения срока хранения копии, воспользовавшись ключом `--expire-time`. Например:

```
pg_probackup set-backup -B каталог_копий --instance имя_экземпляра -
i ид_резервной_копии --expire-time="2020-01-01 00:00:00+03"
```

Также можно воспользоваться ключами `--ttl` и `--expire-time` команды backup и сразу закрепить создаваемую копию:

```
pg_probackup backup -B каталог_копий --instance имя_экземпляра -b FULL --ttl=30d
pg_probackup backup -B каталог_копий --instance имя_экземпляра -b FULL --expire-
time="2020-01-01 00:00:00+03"
```

Определить, закреплена ли копия, можно с помощью команды show:

```
pg_probackup show -B каталог_копий --instance имя_экземпляра -i ид_резервной_копии
```

Если копия закреплена, у неё есть атрибут `expire-time`, содержащий время окончания срока её хранения:

```
...
recovery-time = '2017-05-16 12:57:31'
expire-time = '2020-01-01 00:00:00+03'
data-bytes = 22288792
...
```

Отменить закрепление копии можно, передав в параметре `--ttl` ноль:

```
pg_probackup set-backup -B каталог_копий --instance имя_экземпляра -
i ид_резервной_копии --ttl=0
```

> **Примечание**
>
> Для закреплённой инкрементальной копии неявным образом закрепляются все нужные ей родительские копии. Если вы позже снимите с неё закрепление, последние также автоматически перестанут быть закреплёнными.

Настройка политики хранения архива WAL

Когда осуществляется непрерывное архивирование WAL, сегменты WAL могут занимать много места на диске. Даже если вы время от времени удаляете старые резервные копии, с флагом `--delete-wal` могут быть удалены только те сегменты WAL, которые не относятся ни к какой копии из остающихся в каталоге. Однако если возможность восстановления на момент времени нужна только для последних копий, можно настроить политику хранения архива WAL, чтобы ограничить глубину архива и сэкономить место на диске.

Чтобы настроить политику хранения архива WAL, запустите команду set-config с параметром `--wal-depth`, задающим количество копий, которые могут быть использованы для PITR. Этот параметр действует для всех линий времени, поэтому вы сможете выполнять PITR для одинакового количества копий на каждой линии времени, при их наличии. Закреплённые копии в этом числе не учитываются: если закрепляется одна из последних копий, pg_probackup обеспечивает возможность PITR для каждой дополнительной копии.

Чтобы удалить сегменты WAL, не удовлетворяющие заданной политике хранения архива WAL, вы должны просто выполнить команду delete или backup с флагом `--delete-wal`. Для архивных копий сегменты WAL между `Start LSN` и `Stop LSN` всегда сохраняются, так что эти копии остаются рабочими вне зависимости от значения `--wal-depth` и при необходимости могут быть восстановлены.

Вы также можете использовать параметр `--wal-depth` с командами delete и backup для переопределения ранее заданной политики сохранения WAL и удаления старых сегментов WAL «на лету».

Предположим, что вы заархивировали экземпляр `node` в *каталог_копий* и настроили непрерывное архивирование WAL:

```
pg_probackup show -B каталог_копий --instance node

BACKUP INSTANCE 'node'
============================================================================================================
 Instance  Version  ID      Recovery Time           Mode   WAL Mode  TLI  Time   Data
 WAL  Zratio  Start LSN   Stop LSN    Status
============================================================================================================
 node      11       PZ9442  2019-10-12 10:43:21+03  DELTA  STREAM    1/0   10s  121kB
 16MB    1.00  0/46000028  0/46000160  OK
 node      11       PZ943L  2019-10-12 10:43:04+03  FULL   STREAM    1/0   10s  180MB
 32MB    1.00  0/44000028  0/44000160  OK
 node      11       PZ7YR5  2019-10-11 19:49:56+03  DELTA  STREAM    1/1   10s  112kB
 32MB    1.00  0/41000028  0/41000160  OK
 node      11       PZ7YMP  2019-10-11 19:47:16+03  DELTA  STREAM    1/1   10s  376kB
 32MB    1.00  0/3E000028  0/3F0000B8  OK
 node      11       PZ7YK2  2019-10-11 19:45:45+03  FULL   STREAM    1/0   11s  180MB
 16MB    1.00  0/3C000028  0/3C000198  OK
 node      11       PZ7YFO  2019-10-11 19:43:04+03  FULL   STREAM    1/0   10s   30MB
 16MB    1.00  0/2000028   0/200ADD8   OK
```

Вы можете проверить состояние архива WAL, запустив команду show с ключом `--archive`:

```
pg_probackup show -B каталог_копий --instance node --archive

ARCHIVE INSTANCE 'node'
============================================================================================================
 TLI  Parent TLI  Switchpoint  Min Segno                 Max Segno                 N
 segments  Size  Zratio  N backups  Status
============================================================================================================
 1    0           0/0          000000010000000000000001  000000010000000000000047  71
       36MB  31.00   6          OK
```

Операция очистки WAL без ключа `--wal-depth` может удалить лишь один сегмент:

```
pg_probackup delete -B каталог_копий --instance node --delete-wal

ARCHIVE INSTANCE 'node'
============================================================================================================
 TLI  Parent TLI  Switchpoint  Min Segno                 Max Segno                 N
 segments  Size  Zratio  N backups  Status
============================================================================================================
 1    0           0/0          000000010000000000000002  000000010000000000000047  70
       34MB  32.00   6          OK
```

Например, если вы хотите оставить только те сегменты, которые могут применяться к самой последней копии, передайте в параметре `--wal-depth` значение 1:

```
pg_probackup delete -B каталог_копий --instance node --delete-wal --wal-depth=1

ARCHIVE INSTANCE 'node'
============================================================================================================
 TLI  Parent TLI  Switchpoint  Min Segno                 Max Segno                 N
 segments  Size   Zratio  N backups  Status
============================================================================================================
 1    0           0/0          000000010000000000000046  000000010000000000000047  2
       143kB  228.00  6          OK
```

Также вы можете использовать параметр `--wal-depth` с командой backup:

```
pg_probackup backup -B каталог_данных --instance node -b DELTA --wal-depth=1 --delete-
wal

ARCHIVE INSTANCE 'node'
```

```
==========================================================================================
 TLI  Parent TLI  Switchpoint  Min Segno                 Max Segno                  N
 segments  Size  Zratio  N backups  Status
==========================================================================================
 1    0           0/0          000000010000000000000048  000000010000000000000049  1
       72kB  228.00  7          OK
```

Объединение резервных копий

По мере того как вы будете делать новые и новые инкрементальные копии, общий размер каталога резервных копий может существенно увеличиться. Для экономии места на диске вы можете объединить инкрементальные копии с родительской полной копией, выполнив команду `merge` и передав ей идентификатор копии самой последней резервной копии, подлежащей объединению:

```
pg_probackup merge -B каталог_копий --instance имя_экземпляра -i ид_резервной_копии
```

Эта команда объединяет копии, относящиеся к одной цепочке инкрементальных копий. Если выбирается полная копия, она будет объединена с первой инкрементальной копией после неё. Если выбрана инкрементальная копия, он будет объединена с родительской полной копией, включая все инкрементальные копии между ними. После завершения объединения результирующая полная копия будет вмещать в себя все данные, а инкрементальные копии будут удалены как избыточные. Таким образом, операция объединения по сути равнозначна созданию новой полной копии с удалением всех устаревших копий, но выполняется она быстрее, особенно с большими объёмами данных, и не нагружает подсистему ввода/вывода и сеть (если pg_probackup работает в удалённом режиме).

Перед объединением pg_probackup проверяет все задействуемые резервные копии, чтобы удостовериться в их целостности. Вы можете проверить текущее состояние резервной копии, передав её идентификатор команде show.

```
pg_probackup show -B каталог_копий --instance имя_экземпляра -i ид_резервной_копии
```

Если процесс объединения ещё не закончен, вы увидите состояние `MERGING`. Для полных копий также можно увидеть состояние `MERGED` в процессе изменения метаданных на последнем этапе объединения. В случае прерывания операции объединения она может быть перезапущена.

Удаление резервных копий

Для удаления резервной копии, ставшей ненужной, выполните команду:

```
pg_probackup delete -B каталог_копий --instance имя_экземпляра -i ид_резервной_копии
```

Эта команда удалит резервную копию с заданным *ид_резервной_копии* вместе со всеми инкрементальными копиями, которые от неё зависят (если таковые найдутся). Таким образом вы можете удалить некоторые последние инкрементальные копии, сохранив предыдущую полную копию и некоторые следующие за ней инкрементальные копии.

Для удаления старых файлов WAL, которые не нужны для восстановления никаких из оставшихся резервных копий, воспользуйтесь ключом `--delete-wal`:

```
pg_probackup delete -B каталог_копий --instance имя_экземпляра --delete-wal
```

Чтобы удалить резервные копии, считающиеся ненужными согласно текущей политике хранения, воспользуйтесь ключом `--delete-expired`:

```
pg_probackup delete -B каталог_копий --instance имя_экземпляра --delete-expired
```

Копии с истёкшим сроком нельзя удалить, пока на них базируется минимум одна инкрементальная копия, удовлетворяющая политике хранения. Если вы хотите сократить число резервных копий, требующихся для восстановления инкрементальных копий, укажите параметр `--merge-expired` при выполнении этой команды:

```
pg_probackup delete -B каталог_копий --instance имя_экземпляра --delete-expired --
merge-expired
```

В этом случае pg_probackup ищет самую старую инкрементальную копию, удовлетворяющую политике хранения и объединяет её с базовыми для неё полной и инкрементальными копиями, срок хранения которых истёк, тем самым превращая её в полную копию. После завершения объединения ставшие ненужными копии удаляются.

Прежде чем объединять или удалять резервные копии, вы можете выполнить команду `delete` с параметром `--dry-run` и получить в результате состояние всех имеющихся копий в соответствии с текущей политикой хранения; никакие необратимые действия при этом выполняться не будут.

Для удаления всех копий с определённым состоянием, воспользуйтесь параметром `--status`:

```
pg_probackup delete -B каталог_копий --instance имя_экземпляра --status=ERROR
```

При удалении копий по критерию состояния установленные политики сохранения не учитываются.

Клонирование и синхронизация экземпляра Postgres Pro

В pg_probackup реализована команда catchup, которая позволяет создать копию экземпляра сервера Postgres Pro напрямую, не используя каталог резервных копий. Эта команда может быть полезна:

- Для добавления нового ведомого сервера.

  Как правило, для создания копии экземпляра Postgres Pro используется pg_basebackup. Команда `catchup`, если указан пустой целевой каталог, сделает то же самое, но в параллельном режиме может сработать гораздо быстрее.

- Для синхронизации отставшего ведомого сервера с ведущим.

  При активной записи на ведущем сервере реплики могут не успевать воспроизводить WAL с достаточной скоростью и в результате отставать. Обычно в этом случае создаётся новая реплика, а для этого нужно передать и сохранить на диске большой объём данных. Команда `catchup` напрямую переносит различия с ведущего сервера, что намного быстрее позволяет обновить данные на уже существующей реплике.

Операция `catchup` отличается от других операций pg_probackup:

- Не требуется каталог резервных копий.
- В качестве метода доставки WAL поддерживается только STREAM.
- Копирование внешних каталогов не поддерживается.
- Одновременно с `catchup` нельзя выполнять команды DDL CREATE TABLESPACE/DROP TABLESPACE.
- При синхронизации `catchup` берёт файлы конфигурации, такие как `postgresql.conf`, `postgresql.auto.conf` или `pg_hba.conf`, с исходного сервера и заменяет ими соответствующие файлы на целевом сервере. Чтобы оставить файлы конфигурации без изменений, используйте параметр `--exclude-path`.

Чтобы подготовить экземпляр Postgres Pro к клонированию/синхронизации, настройте исходный сервер БД следующим образом:

- Настройте кластер баз данных для копирования экземпляра сервера.
- Для копирования с удалённого сервера настройте удалённый режим.
- Для использования режима PTRACK настройте копирование PTRACK.

Перед клонированием/синхронизацией экземпляра Postgres Pro убедитесь, что исходный сервер запущен и принимает подключения. Чтобы клонировать/синхронизировать экземпляр Postgres Pro, в системе с целевым сервером выполните команду catchup:

```
pg_probackup catchup -b режим_синхронизации --source-
pgdata=путь_к_копируемому_каталогу_данных --destination-
pgdata=путь_к_целевому_каталогу_данных --stream [параметры_подключения]
 [параметры_удалённого_режима]
```

Здесь `режим_синхронизации` может принимать следующие значения: `FULL`, `PAGE`, `DELTA`, `PTRACK`.

- FULL — создаётся полная копия экземпляра Postgres Pro, для этого целевой каталог данных БД должен быть пустым.
- DELTA — считываются все файлы данных в каталоге данных и создаётся инкрементальная копия для страниц, изменённых с момента остановки целевого экземпляра.
- PTRACK — изменения в страницах отслеживаются на лету, считываются и копируются только страницы, изменённые с точки расхождения исходного и целевого экземпляров.

> **Предупреждение**
>
> Чтобы синхронизировать экземпляр в режиме PTRACK, необходим PTRACK версии 2.0 или выше, а значит, и Postgres Pro версии не ниже 11.

Указав параметр `--stream`, можно задать режим STREAM, при котором все необходимые файлы WAL передаются с исходного сервера по протоколу репликации.

Вы можете задать параметры_подключения к исходной БД, а если она находится на другом сервере, также укажите параметры_удалённого_режима.

Если в исходной БД есть табличные пространства, которые должны располагаться в других каталогах в целевой системе, задайте также параметр `--tablespace-mapping`:

```
pg_probackup catchup -b режим_синхронизации --source-
pgdata=путь_к_копируемому_каталогу_данных --destination-
pgdata=путь_к_целевому_каталогу_данных --stream --tablespace-
mapping=СТАРЫЙ_КАТАЛОГ=НОВЫЙ_КАТАЛОГ
```

Чтобы операция `catchup` выполнялась в несколько параллельных потоков, задайте число потоков с помощью параметра `--threads`:

```
pg_probackup catchup -b режим_синхронизации --source-
pgdata=путь_к_копируемому_каталогу_данных --destination-
pgdata=путь_к_целевому_каталогу_данных --stream --threads=число_потоков
```

Перед клонированием/синхронизацией экземпляра Postgres Pro вы можете запустить команду `catchup` с флагом `--dry-run`, чтобы оценить размер передаваемых файлов данных без внесения изменений на диск:

```
pg_probackup catchup -b режим_синхронизации --source-
pgdata=путь_к_копируемому_каталогу_данных --destination-
pgdata=путь_к_целевому_каталогу_данных --stream --dry-run
```

Например, предположим, что отстал удалённый ведомый экземпляр Postgres Pro, данные которого хранятся в каталоге `/replica-pgdata`. Чтобы синхронизировать этот экземпляр с экземпляром в каталоге данных `/master-pgdata`, вы можете выполнить команду `catchup`, включив режим `PTRACK` и используя четыре параллельных потока, следующим образом:

```
pg_probackup catchup --source-pgdata=/master-pgdata --destination-pgdata=/replica-
pgdata -p 5432 -d postgres -U remote-postgres-user --stream --backup-mode=PTRACK
 --remote-host=remote-hostname --remote-user=remote-unix-username -j 4 --exclude-
path=postgresql.conf --exclude-path=postgresql.auto.conf --exclude-path=pg_hba.conf --
exclude-path=pg_ident.conf
```

Обратите внимание, что в этом примере файлы конфигурации не будут перезаписаны во время синхронизации.

В примере ниже показано, как можно добавить ещё один удалённый ведомый сервер Postgres Pro с каталогом данных `/replica-pga`, запустив `catchup` в режиме `FULL` в четыре параллельных потока:

```
pg_probackup catchup --source-pgdata=/master-pgdata --destination-pgdata=/replica-
pgdata -p 5432 -d postgres -U remote-postgres-user --stream --backup-mode=FULL --
remote-host=remote-hostname --remote-user=remote-unix-username -j 4
```

Справка по командной строке

Команды

В этом подразделе описываются команды pg_probackup. Необязательные параметры этих команд заключаются в квадратные скобки. В подробностях все параметры описываются в подразделе Параметры.

version

```
pg_probackup version
```

Выводит версию pg_probackup.

help

```
pg_probackup help [команда]
```

Выдаёт справку по командам pg_probackup. Если в параметрах задаётся одна из команд pg_probackup, выводит подробную информацию по параметрам, которые принимает эта команда.

init

```
pg_probackup init -B каталог_копий [--help]
```

Инициализирует *каталог_копий*, в котором будут храниться резервные копии, архив WAL и метаинформация о скопированных кластерах баз данных. Если заданный *каталог_копий* уже существует, он должен быть пустым. В противном случае pg_probackup выдаст соответствующее сообщение об ошибке.

За подробностями обратитесь к подразделу Инициализация каталога копий.

add-instance

```
pg_probackup add-instance -B каталог_копий -D каталог_данных --instance имя_экземпляра
 [--help]
```

Инициализирует новый копируемый экземпляр в каталоге *каталог_копий* и создаёт файл конфигурации `pg_probackup.conf`, управляющий параметрами pg_probackup, относящимися к кластеру в указанном *каталоге_данных*.

За подробностями обратитесь к подразделу Определение копируемого экземпляра.

del-instance

```
pg_probackup del-instance -B каталог_копий --instance имя_экземпляра [--help]
```

Удаляет все резервные копии и файлы WAL, связанные с указанным экземпляром.

set-config

```
pg_probackup set-config -B каталог_копий --instance имя_экземпляра
[--help] [--pgdata=путь_к_pgdata]
[--retention-redundancy=избыточность][--retention-window=окно][--wal-depth=глубина_wal]
[--compress-algorithm=алгоритм_сжатия] [--compress-level=уровень_сжатия]
[-d имя_базы] [-h сервер] [-p порт] [-U имя_пользователя]
[--archive-timeout=тайм-аут] [--external-dirs=путь_внешнего_каталога]
[--restore-command=команда]
[параметры_удалённого_режима] [параметры_удалённого_архива_wal] [параметры_журнала]
```

Добавляет заданные параметры соединения, сжатия, хранения, ведения журнала и указания внешних каталогов в конфигурационный файл `pg_probackup.conf` либо изменяет ранее заданные значения.

Все поддерживаемые параметры описываются в подразделе Параметры.

Редактировать `pg_probackup.conf` вручную **не рекомендуется**.

set-backup

```
pg_probackup set-backup -B каталог_копий --instance имя_экземпляра -
i ид_резервной_копии
{--ttl=время_жизни | --expire-time=время}
[--note=заметка_к_копии] [--help]
```

Устанавливает заданные для конкретной резервной копии параметры в конфигурационном файле `backup.control` или изменяет ранее определённые значения.

`--note=`*заметка_к_копии*

Задаёт текстовую заметку для резервной копии. Если *заметка_к_копии* содержит символы перевода строки, сохранена будет только подстрока до первого перевода строки. Максимальный размер заметки равен 1 КБ. Значение *'none'* удаляет текущую заметку.

Все поддерживаемые варианты закрепления описываются в подразделе Параметры закрепления.

show-config

```
pg_probackup show-config -B каталог_копий --instance имя_экземпляра [--format=plain|
json]
```

Выводит содержимое файла `pg_probackup.conf`, размещённого в каталоге *каталог_копий*`/backups/`*имя_экземпляра*. Вы можете добавить параметр `--format=json` для получения результата в формате JSON. По умолчанию параметры конфигурации выводятся обычным текстом.

Чтобы изменить содержимое `pg_probackup.conf`, используйте команду set-config.

show

```
pg_probackup show -B каталог_копий
[--help] [--instance имя_экземпляра [-i ид_резервной_копии | --archive]] [--
format=plain|json] [--no-color]
```

Показывает содержимое каталога копий. Если задано *имя_экземпляра* и *ид_резервной_копии*, выводится подробная информация об этой копии. С указанием `--archive` эта команда показывает содержимое архива WAL в данном каталоге.

По умолчанию содержимое каталога представляется в виде обычного текста. Вы можете передать параметр `--format=json`, чтобы получить результат в формате JSON. С параметром `--no-color` выводимые сообщения не выделяются цветом.

Более подробно использование этой команды описывается в подразделах Управление каталогом резервных копий и Просмотр оглавления архива WAL.

backup

```
pg_probackup backup -B каталог_копий -b режим_копирования --instance имя_экземпляра
[--help] [-j число_потоков] [--progress]
[-C] [--stream [-S slot_name] [--temp-slot]] [--backup-pg-log]
[--no-validate] [--skip-block-validation]
[-w --no-password] [-W --password]
[--archive-timeout=тайм-аут] [--external-dirs=путь_внешнего_каталога]
[--no-sync] [--note=заметка_к_копии]
[параметры_соединения] [параметры_сжатия] [параметры_удалённого_режима]
[параметры_хранения] [параметры_закрепления] [параметры_журнала]
```

Создаёт резервную копию экземпляра Postgres Pro.

`-b` *режим*
`--backup-mode=`*режим*

Выбирает режим резервного копирования. Поддерживаются следующие режимы:

- `FULL` — создаётся полная резервная копия, содержащая все файлы данных кластера, необходимые для его восстановления.
- `DELTA` — считываются все файлы данных в каталоге данных и создаётся инкрементальная копия для страниц, изменённых со времени предыдущего копирования.
- `PAGE` — создаётся инкрементальная копия, содержащая только те страницы, которые фигурируют в файлах WAL, записанных после предыдущей полной или инкрементальной копии.
- `PTRACK` — создаётся инкрементальная резервная копия со страницами, изменения в которых отслеживались на лету.

`-C`
`--smooth-checkpoint`

Растягивает выполнение контрольной точки во времени. По умолчанию pg_probackup пытается произвести контрольную точку максимально быстро.

`--stream`

Создаёт потоковую резервную копию (STREAM), включая в неё все необходимые файлы WAL, получаемые от сервера по протоколу репликации.

`--temp-slot`

Создаёт временный слот физической репликации для передачи WAL с архивируемого экземпляра Postgres Pro. Это гарантирует, что все нужные сегменты WAL будут доступны, если в процессе копирования произойдёт переключение сегментов WAL. Этот параметр может использоваться только вместе с параметром `--stream`. По умолчанию имя слота — `pg_probackup_slot`, но его можно поменять, воспользовавшись параметром `--slot`/`-S`.

`-S` *имя_слота*
`--slot=`*имя_слота*

Задаёт слот репликации для передачи WAL. Этот параметр можно указать только вместе с параметром `--stream`.

`--backup-pg-log`

Включает в резервную копию каталог `log`. Этот каталог обычно содержит журналы сообщений сервера. По умолчанию каталог `log` в копию не включается.

`-E` *путь_внешнего_каталога*
`--external-dirs=`*путь_внешнего_каталога*

Включает в создаваемую копию указанный каталог, рекурсивно копируя его содержимое в отдельный подкаталог каталога резервной копии. Этот параметр полезен для архивирования скриптов, SQL-дампов и файлов конфигурации, расположенных вне каталога данных. Если вы хотите архивировать несколько внешних каталогов, их пути нужно разделять двоеточием в Linux или точкой с запятой в Windows.

`--archive-timeout=`*время_ожидания*

Задаёт тайм-аут для архивирования сегментов WAL и потоковой передачи (в секундах). По умолчанию pg_probackup ждёт выполнения этих операций 300 секунд.

`--skip-block-validation`

Отключает проверку контрольных сумм на уровне блоков в процессе резервного копирования.

`--no-validate`

Пропускает автоматическую проверку созданной резервной копии. Этот ключ может быть полезен, если вы регулярно проверяете резервные копии и хотите сократить время создания копии.

`--no-sync`

Не сбрасывать копируемые файлы на диск. Этот флаг позволяет несколько ускорить процесс копирования. Использование этого флага может привести к повреждению данных в случае аварии операционной системы или аппаратного сбоя. Если вы используете его, рекомендуется выполнить команду validate сразу после завершения копирования, чтобы убедиться в отсутствии проблем.

`--note=`*`заметка_к_копии`*

Задаёт текстовую заметку для резервной копии. Если *`заметка_к_копии`* содержит символы перевода строки, сохранена будет только подстрока до первого перевода строки. Максимальный размер заметки равен 1 КБ. Значение `'none'` удаляет текущую заметку.

Дополнительно вы можете задать параметры соединения, хранения, закрепления, удалённого режима, сжатия и ведения журнала, а также общие параметры.

За подробностями обратитесь к подразделу Создание резервной копии.

restore

```
pg_probackup restore -B каталог_копий --instance имя_экземпляра
[--help] [-D каталог_данных] [-i ид_резервной_копии]
[-j число_потоков] [--progress]
[-T СТАРЫЙ_КАТАЛОГ=НОВЫЙ_КАТАЛОГ] [--external-mapping=СТАРЫЙ_КАТАЛОГ=НОВЫЙ_КАТАЛОГ] [--
skip-external-dirs]
[-R | --restore-as-replica] [--no-validate] [--skip-block-validation]
[--force] [--no-sync]
[--restore-command=команда]
[--primary-conninfo=строка_подключения]
[-S | --primary-slot-name=имя_слота]
[-X каталог_WAL | --waldir=каталог_WAL]
[параметры_точки_восстановления] [параметры_журнала] [параметры_удалённого_режима]
[параметры_частичного_восстановления] [параметры_удалённого_архива_wal]
```

Восстанавливает экземпляр Postgres Pro из резервной копии, расположенной в каталоге *`каталог_копий`*. Если вы укажете один или несколько параметров точки восстановления, pg_probackup найдёт ближайшую к этой точке копию и выполнит восстановление до заданной точки. Если ни идентификатор копии, ни параметры точки восстановления не задаются, pg_probackup выполняет восстановление, используя самую последнюю копию.

`-R`
`--restore-as-replica`

Создаёт минимальный файл конфигурации восстановления для облегчения настройки ведомого сервера. Если для соединения репликации требуется пароль, его нужно дополнительно задать вручную в primary_conninfo. Для Postgres Pro версии 11 и ниже параметры восстановления сохраняются в каталоге данных в файле `recovery.conf`, но для версий Postgres Pro, начиная с 12, pg_probackup сохраняет эти параметры в файле `probackup_recovery.conf`.

`--primary-conninfo=`*`строка_подключения`*

Устанавливает заданное значение для параметра primary_conninfo. Это значение учитывается только при использовании флага `-R`.

Пример: `--primary-conninfo="host=192.168.1.50 port=5432 user=foo password=foopass"`

`-S`
`--primary-slot-name=`*`имя_слота`*

Устанавливает заданное значение для параметра primary_slot_name. Это значение учитывается только при использовании флага `-R`.

`-T` *`СТАРЫЙ_КАТАЛОГ=НОВЫЙ_КАТАЛОГ`*
`--tablespace-mapping=`*`СТАРЫЙ_КАТАЛОГ=НОВЫЙ_КАТАЛОГ`*

Перемещает табличное пространство из каталога *`СТАРЫЙ_КАТАЛОГ`* в *`НОВЫЙ_КАТАЛОГ`* во время восстановления. И *`СТАРЫЙ_КАТАЛОГ`*, и *`НОВЫЙ_КАТАЛОГ`* должны задаваться абсолютными путями. Если путь содержит знак равно (=), экранируйте этот знак обратной косой чертой. Данный параметр может указываться неоднократно для перемещения нескольких табличных пространств.

`--external-mapping=`*`СТАРЫЙ_КАТАЛОГ=НОВЫЙ_КАТАЛОГ`*

Перемещает внешний каталог, включённый в резервную копию, из каталога *`СТАРЫЙ_КАТАЛОГ`* в *`НОВЫЙ_КАТАЛОГ`* во время восстановления. И *`СТАРЫЙ_КАТАЛОГ`*, и *`НОВЫЙ_КАТАЛОГ`* должны задаваться абсолютными путями. Если путь содержит знак равно (=), экранируйте этот знак обратной косой чертой. Данный параметр может указываться неоднократно для нескольких каталогов.

`--skip-external-dirs`

Пропускать внешние каталоги, включённые в резервную копию указанием `--external-dirs`. Содержимое этих каталогов не будет восстановлено.

`--skip-block-validation`

Отключает проверку контрольных сумм на уровне блоков для ускорения проверки целостности. При автоматической проверке перед восстановлением будут проверяться только контрольные суммы на уровне файлов.

`--no-validate`

Пропускает проверку резервного копирования. Этот ключ может быть полезен, если вы регулярно проверяете копии и хотите сократить время восстановления данных.

`--restore-command=`*`команда`*

Задаёт значение для параметра restore_command. Например: `--restore-command='cp /mnt/server/archivedir/%f "%p"'`

`--force`

Позволяет игнорировать ошибочное состояние копии. Этот флаг можно использовать, когда требуется восстановить кластер Postgres Pro из повреждённой или некорректной копии. Применяйте его с осторожностью. Если в целевом каталоге `PGDATA` идентификатор системы отличается от того, что указан в копии, при инкрементальном восстановлении с этим флагом содержимое каталога будет перезаписано (тогда как без флага возникнет ошибка). Также в случае перенаправления указанием `--tablespace-mapping` табличных пространств в непустые каталоги содержимое этих каталогов будет удалено.

`--no-sync`

Не сбрасывать восстанавливаемые файлы на диск. Этот флаг позволяет несколько ускорить процесс восстановления. Использование этого флага может привести к повреждению данных в случае аварии операционной системы или аппаратного сбоя. Если такое событие произойдёт, вам потребуется запустить команду restore ещё раз.

`-X` *`каталог_WAL`*
`--waldir=`*`каталог_WAL`*

Указывает каталог, в котором должен храниться WAL.

Дополнительно вы можете задать параметры точки восстановления, удалённого сервера, удалённого архива WAL, ведения журнала и частичного восстановления, а также общие параметры.

За подробностями обратитесь к подразделу Восстановление кластера.

checkdb

```
pg_probackup checkdb
[-B каталог_копий] [--instance имя_экземпляра] [-D каталог_данных]
[--help] [-j число_потоков] [--progress]
[--amcheck [--skip-block-validation] [--checkunique] [--heapallindexed]]
[параметры_соединения] [параметры_журнала]
```

Проверяет целостность кластера баз данных Postgres Pro, выявляя физические и логические повреждения.

`--amcheck`

Производит логическую проверку индексов для указанного экземпляра Postgres Pro, если не были выявлены повреждения при проверке файлов данных. Для проверки индексов в базе данных в ней должно быть установлено расширение amcheck или amcheck_next. В базах данных, где amcheck отсутствует, индексы проверяться не будут. Дополнительные параметры `--checkunique` и `--heapallindexed` действуют в зависимости от установленной версии amcheck.

`--checkunique`

Проверяет ограничения уникальности во время логической проверки индексов. Вы можете использовать этот флаг только вместе с флагом `--amcheck`, когда в базе данных установлено расширение amcheck.

Эта проверка возможна, только если в используемой вами версии расширения amcheck функция `bt_index_check` принимает параметр *checkunique*.

`--heapallindexed`

Проверяет, отражены ли в индексах все кортежи кучи, которые должны быть проиндексированы. Этот ключ можно использовать только вместе с `--amcheck`.

Эта проверка возможна, только если в используемой вами версии расширения amcheck/ amcheck_next функция `bt_index_check` принимает параметр *heapallindexed*.

`--skip-block-validation`

Пропустить проверку файлов данных. Вы можете использовать этот ключ вместе с `--amcheck`, чтобы произвести только логическую проверку индексов.

Также вы можете задать параметры соединения и ведения журнала.

За подробностями обратитесь к подразделу Проверка кластера.

validate

```
pg_probackup validate -B каталог_копий
[--help] [--instance имя_экземпляра] [-i ид_резервной_копии]
[-j число_потоков] [--progress]
[--skip-block-validation]
[параметры_точки_восстановления] [параметры_журнала]
```

Проверяет наличие и целостность всех файлов, необходимых для восстановления кластера. Если *имя_экземпляра* не задаётся, pg_probackup проверяет все резервные копии, имеющиеся в каталоге копий. Если вы зададите *имя_экземпляра* без дополнительных параметров, pg_probackup проверит все резервные копии, которые имеются для этого экземпляра. Если задать *имя_экземпляра* с указанием точки восстановления или *ид_резервной_копии*, pg_probackup проверит, возможно ли восстановить кластер с этими параметрами.

За подробностями обратитесь к подразделу Проверка копии.

merge

```
pg_probackup merge -B каталог_копий --instance имя_экземпляра -i ид_резервной_копии
[--help] [-j число_потоков]  [--progress] [--no-validate] [--no-sync]
[параметры_журнала]
```

Объединяет копии, относящиеся к одной цепочке инкрементальных копий. Если выбрана полная копия, она будет объединена с первой инкрементальной копией после неё. Если выбрана инкрементальная копия, она будет объединена с родительской полной копией, включая все инкрементальные копии между ними. После завершения объединения результирующая полная копия будет вмещать в себя все данные, а инкрементальные копии будут удалены как избыточные.

`--no-validate`

Пропускает автоматическую проверку до и после объединения.

`--no-sync`

Не сбрасывать объединяемые файлы на диск. Этот флаг позволяет несколько ускорить процесс объединения. Использование этого флага может привести к повреждению данных в случае аварии операционной системы или аппаратного сбоя.

За подробностями обратитесь к подразделу Объединение резервных копий.

delete

```
pg_probackup delete -B каталог_копий --instance имя_экземпляра
[--help] [-j число_потоков] [--progress]
[--retention-redundancy=избыточность][--retention-window=окно][--wal-depth=глубина_wal]
 [--delete-wal]
{-i ид_резервной_копии | --delete-expired [--merge-expired] | --merge-expired | --
status=состояние}
[--dry-run] [--no-validate] [--no-sync] [параметры_журнала]
```

Удаляет копию с заданным идентификатором (*ид_резервной_копии*) или запускает процедуру удаления резервных копий или заархивированных файлов WAL, не удовлетворяющих текущей политике хранения.

`--no-validate`

Пропускает автоматическую проверку до и после объединения копий в соответствии с политикой хранения.

`--no-sync`

Не сбрасывать объединяемые файлы на диск. Этот флаг позволяет несколько ускорить процесс объединения копий в соответствии с политикой хранения. Использование этого флага может привести к повреждению данных в случае аварии операционной системы или аппаратного сбоя.

За подробностями обратитесь к подразделам Удаление резервных копий, Параметры хранения и Настройка политики хранения.

archive-push

```
pg_probackup archive-push -B каталог_копий --instance имя_экземпляра
--wal-file-name=имя_файла_wal [--wal-file-path=путь_файлов_wal]
[--help] [--no-sync] [--compress] [--no-ready-rename] [--overwrite]
[-j число_потоков] [--batch-size=размер_порции]
[--archive-timeout=тайм-аут]
[--compress-algorithm=алгоритм_сжатия]
[--compress-level=уровень_сжатия]
[параметры_удалённого_режима] [параметры_журнала]
```

Копирует файлы WAL в соответствующий подкаталог каталога копий, проверяя целевой экземпляр по *имени_экземпляра* и значению `system-identifier`. Если параметры экземпляра резервной копии и кластера не совпадают, операция копирования не выполняется, и выдаётся ошибка: `Refuse to push WAL segment segment_name into archive. Instance parameters mismatch.` (Отказано в помещении сегмента имя_сегмента в архив. Параметры экземпляра не совпадают.)

Если файлы, которые требуется копировать, уже имеются в каталоге копий, pg_probackup вычисляет и сравнивает их контрольные суммы. В случае совпадения контрольных сумм `archive-push` пропускает соответствующий файл и выдаёт код успешного завершения. Если же они не совпадают, операция `archive-push` завершается с ошибкой. Если вы хотите заменять существующие файлы WAL в случае несовпадения контрольных сумм, добавьте к команде `archive-push` ключ `--overwrite`.

Содержимое каждого файла копируется во временный файл с расширением `.part`. Если такой временный файл уже существует, pg_probackup ждёт, что он исчезнет в течение заданного параметром `archive_timeout` времени, а если этого не происходит, отбрасывает его. После переноса содержимого выполняется атомарная операция переименования. Тем самым гарантируется, что в случае ошибки команды `archive-push` непрерывное архивирование не остановится и что при параллельном архивировании WAL из разных источников в один архив повреждение архива исключено.

Для ускорения архивирования вы можете воспользоваться параметром `--batch-size`, определяющим размер порции из нескольких копируемых сегментов WAL. Вместе с параметром `--batch-size` также можно применить указание `-j`, чтобы порции копировались в несколько потоков.

Сегменты WAL, копируемые в архив, по умолчанию (без указания флага `--no-sync`) гарантированно сбрасываются на диск.

Команду `archive-push` можно использовать в значении параметра archive_command Postgres Pro при настройке непрерывного архивирования WAL.

За подробностями обратитесь к подразделам Параметры архивирования и Параметры сжатия.

archive-get

```
pg_probackup archive-get -B каталог_копий --instance имя_экземпляра --wal-file-
path=путь_файлов_wal --wal-file-name=имя_файла_wal
[-j число_потоков] [--batch-size=размер_порции]
[--prefetch-dir=каталог_предвыборки] [--no-validate-wal]
[--help] [параметры_удалённого_режима] [параметры_журнала]
```

Копирует файлы WAL из соответствующего подкаталога каталога резервных копий в каталог журнала предзаписи кластера. Эта команда автоматически устанавливается программой pg_probackup в значении параметра `restore_command` при восстановлении архивных копий с применением архива WAL. Устанавливать её вручную не нужно.

Для ускорения восстановления вы можете воспользоваться параметром `--batch-size`, определяющим размер порции из нескольких копируемых сегментов WAL. Вместе с параметром `--batch-size` также можно применить указание `-j`, чтобы порции сегментов копировались в несколько потоков.

За подробностями обратитесь к подразделу Параметры архивации.

catchup

```
pg_probackup catchup -b режим_синхронизации
--source-pgdata=путь_к_копируемому_каталогу_данных
--destination-pgdata=путь_к_целевому_каталогу_данных
[--help] [-j | --threads=число_потоков] [--stream] [--dry-run]
[--temp-slot] [-P | --perm-slot] [-S | --slot=имя_слота]
[--exclude-path=ПУТЬ]
```

[-T *СТАРЫЙ_КАТАЛОГ=НОВЫЙ_КАТАЛОГ*]
[*параметры_подключения*] [*параметры_удалённого_режима*]

Создаёт копию экземпляра Postgres Pro, не используя каталог резервных копий.

-b *режим_синхронизации*
--backup-mode=*режим_синхронизации*

Выбирает режим синхронизации. Поддерживаются следующие режимы:

- FULL — создаётся полная копия экземпляра Postgres Pro.
- DELTA — считываются все файлы данных в каталоге данных и создаётся инкрементальная копия для страниц, изменённых с момента остановки целевого экземпляра.
- PTRACK — изменения в страницах отслеживаются на лету, считываются и копируются только страницы, изменённые с точки расхождения исходного и целевого экземпляров.

> **Предупреждение**
>
> Чтобы синхронизировать экземпляр в режиме PTRACK, необходим PTRACK версии 2.0 или выше, а значит, и Postgres Pro версии не ниже 11.

--source-pgdata=*путь_к_копируемому_каталогу_данных*

Задаёт путь к каталогу данных копируемого экземпляра. Каталог может находиться как локально, так и удалённо.

--destination-pgdata=*путь_к_целевому_каталогу_данных*

Задаёт путь к локальному целевому каталогу данных.

-j *число_потоков*
--threads=*число_потоков*

Задаёт число параллельных потоков для процесса catchup.

--stream

Копирует экземпляр в режиме доставки STREAM, при котором все необходимые файлы WAL передаются с исходного сервера по протоколу репликации.

--dry-run

Отображает общий размер файлов, которые будут переданы командой catchup. Если установлен флаг --dry-run, выполняется пробный запуск catchup, при котором не создаются, не удаляются и не перемещаются файлы на диске и пропускается потоковая трансляция WAL. Этот флаг также позволяет проверить правильность всех параметров и готовность к запуску клонирования/синхронизации.

-x=*префикс_пути*
--exclude-path=*префикс_пути*

Определяет префикс для файлов, которые не будут копироваться при синхронизации экземпляров Postgres Pro. Такой префикс должен содержать путь относительно каталога данных экземпляра. Если в префиксе указан каталог, ни один файл в этом каталоге не будет копирован.

> **Предупреждение**
>
> Используйте этот параметр с осторожностью, поскольку исключение файлов может привести к неполной синхронизации.

`--temp-slot`

Создаёт *временный* слот физической репликации для передачи WAL с копируемого экземпляра Postgres Pro. Это гарантирует, что все нужные сегменты WAL будут доступны, если в процессе копирования произойдёт переключение сегментов WAL. Этот параметр можно использовать только вместе с параметром `--stream` и нельзя использовать с параметром `--perm-slot`. По умолчанию имя временного слота — `pg_probackup_slot`, но его можно поменять, воспользовавшись параметром `--slot`/`-S`.

`-P`
`--perm-slot`

Создаёт *постоянный* слот физической репликации для передачи WAL с копируемого экземпляра Postgres Pro. Этот параметр можно использовать только вместе с параметром `--stream` и нельзя использовать с параметром `--temp-slot`. По умолчанию имя постоянного слота — `pg_probackup_perm_slot`, но его можно поменять, воспользовавшись параметром `--slot`/`-S`.

`-S` *имя_слота*
`--slot=`*имя_слота*

Задаёт слот репликации для передачи WAL. Этот параметр можно указать только вместе с параметром `--stream`.

`-T` *СТАРЫЙ_КАТАЛОГ=НОВЫЙ_КАТАЛОГ*
`--tablespace-mapping=`*СТАРЫЙ_КАТАЛОГ=НОВЫЙ_КАТАЛОГ*

Перемещает табличное пространство из каталога *СТАРЫЙ_КАТАЛОГ* в *НОВЫЙ_КАТАЛОГ* во время восстановления. И *СТАРЫЙ_КАТАЛОГ*, и *НОВЫЙ_КАТАЛОГ* должны задаваться абсолютными путями. Если путь содержит знак равно (=), экранируйте этот знак обратной косой чертой. Данный параметр может указываться неоднократно для перемещения нескольких табличных пространств.

Также вы можете задать параметры соединения и параметры удалённого режима.

За подробностями обратитесь к подразделу Клонирование и синхронизация экземпляра Postgres Pro.

Параметры

В этом подразделе описываются параметры командной строки для команд pg_probackup. Если какое-либо значение параметра может быть получено из переменной окружения, имя этой переменной указывается в верхнем регистре ниже параметра командной строки. Некоторые значения могут быть получены из файла конфигурации `pg_probackup.conf`, находящегося в каталоге копий.

За подробностями обратитесь к Подразделу «Настройка pg_probackup».

Если некоторый параметр задаётся несколькими способами, значение в командной строке имеет наивысший приоритет, а значение в `pg_probackup.conf` — наименьший.

Общие параметры

Ниже приведён список параметров общего характера.

`-B` *каталог*
`--backup-path=`*каталог*
`BACKUP_PATH`

Задаёт абсолютный путь к каталогу копий. Каталог копий — это каталог, в котором хранятся все файлы резервных копий и метаинформация. Поскольку это расположение необходимо задавать почти для всех команд pg_probackup, имеет смысл указать его один раз в переменной окружения `BACKUP_PATH`. В этом случае каждый раз указывать этот путь в командной строке не нужно.

`-D` *каталог*
`--pgdata=`*каталог*
`PGDATA`

Задаёт абсолютный путь к каталогу данных кластера. Этот параметр является обязательным только для команды add-instance. Другие команды могут получать этот путь из переменной окружения `PGDATA` или из файла конфигурации `pg_probackup.conf`.

`-i` *ид_резервной_копии*
`--backup-id=`*ид_резервной_копии*

Задаёт уникальный идентификатор резервной копии.

`-j` *число_потоков*
`--threads=`*число_потоков*

Задаёт число параллельных потоков, запускаемых командами `backup`, `restore`, `merge`, `validate`, `checkdb` и `archive-push`.

`--progress`

Включает вывод прогресса выполнения операций.

`--help`

Выводит подробную информацию по параметрам, которые принимает эта команда.

Параметры точки восстановления

Если настроено непрерывное архивирование WAL, вы можете передать один из этих параметров с командой restore или validate, чтобы указать момент, до которого должен быть восстановлен или проверен кластер баз данных.

`--recovery-target=immediate|latest`

Определяет, когда остановить восстановление:

- Со значением `immediate` восстановление завершается сразу после достижения согласованного состояния выбранной копии, либо последней копии из имеющихся, если параметр `-i`/`--backup-id` опущен. Такое поведение по умолчанию применяется для копий типа STREAM.
- Со значением `latest` восстановление продолжается до тех пор, пока не будут применены все имеющиеся в архиве сегменты WAL. Такое поведение по умолчанию применяется для копий типа ARCHIVE.

`--recovery-target-timeline=`*линия_времени*

Выбирает линию времени для восстановления. По умолчанию выбирается линия времени указанной резервной копии.

`--recovery-target-lsn=`*lsn*

Указывает, до какого последовательного номера в журнале предзаписи должно производиться восстановление. Может использоваться только при восстановлении кластеров версии 10 или выше.

`--recovery-target-name=`*имя_цели_восстановления*

Указывает именованную точку сохранения, вплоть до которой будет восстановлен кластер.

`--recovery-target-time=`*время*

Указывает точку времени, вплоть до которой будет производиться восстановление. Если часовой пояс не указывается, подразумевается местное время.

Например: `--recovery-target-time="2020-01-01 00:00:00+03"`

`--recovery-target-xid=`*`ид_транзакции`*

Указывает идентификатор транзакции, вплоть до которой будет производиться восстановление.

`--recovery-target-inclusive=`*`логическое_значение`*

Указывает на необходимость остановки сразу после (`true`) либо до (`false`) достижения целевой точки. Этот параметр можно использовать только вместе с параметром `--recovery-target-name`, `--recovery-target-time`, `--recovery-target-lsn` или `--recovery-target-xid`. Значение по умолчанию определяется параметром recovery_target_inclusive.

`--recovery-target-action=pause|promote|shutdown`

Задаёт действие (recovery_target_action), которое должен выполнить сервер по достижении цели восстановления.

По умолчанию: `pause`

Параметры сохранения

Эти параметры могут использоваться с командами backup и delete.

Подробнее о политике хранения рассказывается в подразделе Настройка политики хранения.

`--retention-redundancy=`*`избыточность`*

Указывает, сколько полных резервных копий должно сохраняться в каталоге данных. Должно быть неотрицательным целым числом. Ноль отключает сохранение.

По умолчанию: `0`

`--retention-window=`*`окно`*

Количество дней, в течение которого возможно восстановление. Должно быть неотрицательным целым числом. При нулевом значении окно восстановления отсутствует.

По умолчанию: `0`

`--wal-depth=`*`глубина_wal`*

Количество резервных копий на каждой линии времени, которое должно сохраняться для обеспечения возможности восстановления PITR. Должно быть неотрицательным целым числом. При нулевом значении этот параметр отключается.

По умолчанию: `0`

`--delete-wal`

Удаляет файлы WAL, которые не являются необходимыми для восстановления кластера из имеющихся резервных копий.

`--delete-expired`

Удаляет резервные копии, не удовлетворяющие политике сохранения, определённой в файле конфигурации `pg_probackup.conf`.

`--merge-expired`

Объединяет самую старую инкрементальную копию, удовлетворяющую требованиям политики хранения, с её родительскими копиями, срок хранения которых истёк.

`--dry-run`

Выводит текущее состояние всех имеющихся резервных копий, но никакие операции, например, удаление или объединение старых копий, при этом не производятся.

Параметры закрепления

Эти параметры могут использоваться с командами backup и set-backup.

За подробностями обратитесь к подразделу Закрепление резервных копий.

`--ttl=`*`время_жизни`*

Задаёт время, на которое закрепляется резервная копия. Значение должно быть неотрицательным целым. Нулевое значение отменяет установленное ранее закрепление резервной копии. Поддерживаются следующие единицы измерения: ms (миллисекунды), s (секунды), min (минуты), h (часы), d (дни). По умолчанию подразумеваются секунды.

Например: `--ttl=30d`

`--expire-time=`*`время`*

Определяет момент времени, до которого будет храниться резервная копия. Время должно задаваться в формате ISO-8601. Если часовой пояс не указывается, подразумевается местное время.

Например: `--expire-time="2020-01-01 00:00:00+03"`

Параметры ведения журнала

Эти параметры могут использоваться с любой командой.

`--no-color`

Отключает цветовое выделение сообщений уровней `warning` и `error` в консоли.

`--log-level-console=`*`уровень_сообщений`*

Управляет уровнем сообщений, которые будут выводиться в журнал консоли. Допустимые уровни: `verbose`, `log`, `info`, `warning`, `error` и `off`. Каждый уровень включает все последующие, и с каждым последующим уровнем объём сообщений уменьшается. Вариант `off` отключает вывод в журнал консоли.

По умолчанию: `info`

> **Примечание**
>
> Все выводимые в консоль сообщения журнала передаются через stderr, чтобы их можно было отделить от вывода команд show и show-config.

`--log-level-file=`*`уровень_сообщений`*

Управляет уровнем сообщений, которые будут выводиться в файл журнала. Допустимые уровни: `verbose`, `log`, `info`, `warning`, `error` и `off`. Каждый уровень включает все последующие, и с каждым последующим уровнем объём сообщений уменьшается. Вариант `off` отключает вывод в файл журнала.

По умолчанию: `off`

`--log-filename=`*`файл_журнала`*

Определяет имена для создаваемых файлов журналов. Имена файлов обрабатываются по шаблону `strftime`, так что вы можете использовать спецкоды с % для выбора имён файлов, зависящих от времени.

По умолчанию: `pg_probackup.log`

Например, если задать шаблон `pg_probackup-%u.log`, pg_probackup будет записывать журнал в отдельные файлы по дням недели, и символы `%u` в имени будут заменяться соответствующим

десятичным номером: `pg_probackup-1.log` в понедельник, `pg_probackup-2.log` во вторник и т. д.

Этот параметр действует, если включена запись в журнал (параметром `--log-level-file`).

`--error-log-filename=`*`файл_журнала_событий`*

Определяет имена только для файлов журналов ошибок. Имена файлов обрабатываются по шаблону `strftime`, так что вы можете использовать спецкоды с % для выбора имён файлов, зависящих от времени.

По умолчанию: none

Например, если задать шаблон `error-pg_probackup-%u.log`, pg_probackup будет записывать журнал в отдельные файлы по дням недели, и символы `%u` в имени будут заменяться соответствующим десятичным номером: `error-pg_probackup-1.log` в понедельник, `error-pg_probackup-2.log` во вторник и т. д.

Этот параметр полезен для диагностики и решения возникающих проблем.

`--log-directory=`*`каталог_журнала`*

Определяет каталог, в котором будут создаваться файлы журналов. Вы должны задать в этом параметре абсолютный путь. Этот каталог создаётся только при необходимости, когда в журнал выводится первое сообщение.

По умолчанию: `$BACKUP_PATH/log/`

`--log-format-console=`*`формат_журнала`*

Определяет формат журнала консоли. Устанавливается только из командной строки. Обратите внимание, что этот параметр нельзя указать в файле конфигурации `pg_probackup.conf` посредством команды set-config и что команда backup также воспринимает указание этого параметра в конфигурационном файле как ошибку. Этот параметр может иметь следующие значения:

- Если `plain`, то журнал выводится на консоль в формате обычного текста.
- Если `json`, то журнал выводится на консоль в формате JSON.

По умолчанию: `plain`

`--log-format-file=`*`формат_журнала`*

Определяет используемый формат файлов журнала. Этот параметр может иметь следующие значения:

- Если `plain`, то файлы журнала записываются в формате обычного текста.
- Если `json`, то файлы журнала записываются в формате JSON.

По умолчанию: `plain`

`--log-rotation-size=`*`размер_журнала_для_ротации`*

Максимальный размер отдельного файла журнала. Если это значение достигается, файл журнала прокручивается при выполнении какой-либо команды pg_probackup, за исключением `help` и `version`. Нулевое значение отключает прокрутку в зависимости от размера. Поддерживаются следующие единицы: kB (по умолчанию), MB, GB, TB.

По умолчанию: `0`

`--log-rotation-age=`*`возраст_журнала_для_ротации`*

Максимальное время жизни отдельного файла журнала. Если это значение достигается, файл журнала прокручивается при выполнении какой-либо команды pg_probackup, за исключением

help и version. Время создания последнего файла журнала сохраняется в $BACKUP_PATH/log/log_rotation. Нулевое значение отключает прокрутку по времени. Поддерживаемые единицы: ms (миллисекунды), s (секунды), min (минуты, по умолчанию), h (часы), d (дни).

По умолчанию: 0

Параметры подключения

Эти параметры могут использоваться с командами backup, catchup и checkdb.

pg_probackup поддерживает все переменные окружения libpq.

-d *имя_бд*
--pgdatabase=*имя_бд*
PGDATABASE

Задаёт имя базы данных для подключения. Это подключение используется только для управления процессом резервного копирования, так что вы можете подключиться к любой существующей базе. Если этот параметр не задаётся в командной строке, переменной окружения PGDATABASE или в конфигурационном файле pg_probackup.conf, pg_probackup принимает в качестве имени базы значение переменной окружения PGUSER или имя текущего пользователя, если переменная PGUSER не задана.

-h *сервер*
--pghost=*сервер*
PGHOST

Указывает имя системы, в которой работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

По умолчанию: localhost

-p *порт*
--pgport=*порт*
PGPORT

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

По умолчанию: 5432

-U *имя_пользователя*
--pguser=*имя_пользователя*
PGUSER

Имя пользователя для подключения.

-w
--no-password

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл .pgpass или переменная окружения PGPASSWORD, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

-W
--password

Запрашивать пароль. (Устаревший параметр.)

Параметры сжатия

Эти параметры могут использоваться с командами backup и archive-push.

`--compress-algorithm=`*`алгоритм_сжатия`*

Определяет алгоритм, который будет использоваться для сжатия файлов данных. Возможные значения: `zlib`, `pglz` и `none`. Значение `zlib` или `pglz` включает сжатие. По умолчанию сжатие отключено. Для команды archive-push алгоритм сжатия `pglz` не поддерживается.

По умолчанию: `none`

`--compress-level=`*`уровень_сжатия`*

Определяет уровень сжатия (от 0 до 9, где 0 обозначает отсутствие сжатия, а 9 — наилучшее). Этот параметр может использоваться только вместе с `--compress-algorithm`.

По умолчанию: `1`

`--compress`

Альтернативный вариант одновременного указания `--compress-algorithm=zlib` и `--compress-level=1`.

Параметры архивации

Эти параметры могут использоваться с командой archive-push в значении параметра archive_command и с командой archive-get в значении restore_command.

Дополнительно вы можете задать параметры удалённого сервера и ведения журнала.

`--wal-file-path=`*`путь_файлов_wal`*

Задаёт путь файла WAL в *`archive_command`* и *`restore_command`*. В качестве значения для данного параметра укажите `%p` или явно задайте путь к файлу вне каталога данных. Если этот параметр не задан, используется путь, заданный в файле `pg_probackup.conf`.

`--wal-file-name=`*`имя_файла_wal`*

Задаёт имя файла WAL в *`archive_command`* и *`restore_command`*. В качестве значения для данного параметра укажите `%f` для правильной его обработки. Если значением параметра `--wal-file-path` является путь вне каталога данных, следует явно указывать имя файла.

`--overwrite`

Разрешает перезапись файлов WAL в архиве. Этот ключ действует при выполнении команды archive-push, когда указанный подкаталог каталога резервных копий уже содержит данный файл WAL, и его нужно заменить новой копией. Без этого ключа `archive-push` сообщит, что сегмент WAL уже существует, и прервёт операцию. Если заменяемый файл не изменился, `archive-push` пропускает его, независимо от указания `--overwrite`.

`--batch-size=`*`размер_порции`*

Задаёт максимальное количество файлов, которое может быть скопировано в архив одним процессом `archive-push` или из архива одним процессом `archive-get`.

`--archive-timeout=`*`время_ожидания`*

Задаёт интервал, по истечении которого существующие файлы `.part` будут считаться потерянными. По умолчанию pg_probackup ждёт исчезновения этих файлов 300 секунд. Этот параметр можно использовать только с командой archive-push.

`--no-ready-rename`

Предотвращает переименование файлов состояния в каталоге `archive_status`. Этот параметр полезен, только когда в *`archive_command`* задано несколько команд, и применять его можно только с командой archive-push.

`--no-sync`

Не сбрасывать копируемые файлы WAL на диск. Этот флаг позволяет несколько ускорить процесс архивации. Использование этого флага может привести к повреждению архива WAL в случае аварии операционной системы или аппаратного сбоя. Данный параметр можно указать только с командой archive-push.

`--prefetch-dir=`*`путь`*

Каталог, в котором будут храниться предзагружаемые сегменты WAL при использовании параметра `--batch-size`. Этот каталог должен находиться в той же файловой системе и ниже той же точки монтирования, что и `PGDATA/pg_wal`. По умолчанию эти сегменты размещаются в каталоге `PGDATA/pg_wal/pbk_prefetch`. Данный параметр может применяться только с командой archive-get.

`--no-validate-wal`

Отключает проверку предзагруженных файлов WAL перед использованием. Применяйте этот параметр, если вы хотите увеличить скорость восстановления. Данный параметр может применяться только с командой archive-get.

Параметры удалённого режима

В этом подразделе описываются параметры, связанные с удалённым выполнением операций pg_probackup по SSH. Эти параметры могут использоваться с командами add-instance, set-config, backup, catchup, restore, archive-push и archive-get.

Подробнее о настройке и использовании удалённого режима рассказывается в Подразделе «Настройка удалённого режима» и Подразделе «Использование pg_probackup в удалённом режиме».

`--remote-proto=`*`протокол`*

Задаёт протокол для удалённого выполнения операций. В настоящее время поддерживается только протокол SSH. Возможные значения:

- `ssh` — включает удалённый режим с использованием SSH. Этот вариант используется по умолчанию.
- `none` — явным образом отключает удалённый режим.

Этот параметр можно не задавать, если указывается `--remote-host`.

`--remote-host=`*`целевой_адрес`*

Задаёт имя или IP-адрес целевого удалённого сервера.

`--remote-port=`*`порт`*

Задаёт целевой порт на удалённом сервере.

По умолчанию: `22`

`--remote-user=`*`имя_пользователя`*

Задаёт имя пользователя на удалённом сервере для SSH-соединения. В отсутствие этого параметра используется имя текущего пользователя, устанавливающего SSH-соединения.

`--remote-path=`*`путь`*

Задаёт каталог, в котором pg_probackup установлен на удалённой системе.

`--ssh-options=`*`параметры_ssh`*

Задаёт строку параметров командной строки для SSH. Например, следующим образом можно установить свойства *keep-alive* для SSH-подключений, которые будет открывать pg_probackup: `--ssh-options="-o ServerAliveCountMax=5 -o ServerAliveInterval=60"`. Полный список всех параметров можно найти в *руководстве по ssh_config*.

Параметры удалённого архива WAL

В этом подразделе описываются параметры, позволяющие задать аргументы archive-get для использования удалённого режима в команде restore_command при восстановлении PITR или восстановлении копий типа ARCHIVE.

`--archive-host=`*`целевой_адрес`*

Задаёт значение аргумента `--remote-host` для команды `archive-get`.

`--archive-port=`*`порт`*

Задаёт значение аргумента `--remote-port` для команды `archive-get`.

По умолчанию: `22`

`--archive-user=`*`имя_пользователя`*

Задаёт значение аргумента `--remote-user` для команды `archive-get`. В отсутствие этого указания используется имя пользователя, запускающего кластер Postgres Pro.

По умолчанию: пользователь Postgres Pro

Параметры инкрементального восстановления

В этом подразделе описываются параметры, связанные с инкрементальным восстановлением кластера. Эти параметры могут передаваться с командой restore.

`-I` *`инкрементальный_режим`*
`--incremental-mode=`*`инкрементальный_режим`*

Выбирает инкрементальный режим. Поддерживаются следующие режимы:

- `CHECKSUM` — заменять только страницы с неподходящей контрольной суммой и LSN.
- `LSN` — заменять только те страницы, LSN которых больше точки расхождения.
- `NONE` — обычное восстановление.

Параметры частичного восстановления

В этом подразделе описываются параметры, связанные с частичным восстановлением кластера. Эти параметры могут передаваться с командой restore.

`--db-exclude=`*`имя_бд`*

Задаёт имя базы данных, которая должна быть исключена из числа восстанавливаемых. Все остальные базы данных в кластере будут восстанавливаться, включая `template0` и `template1`. Этот параметр можно задать несколько раз, таким образом исключив несколько баз данных.

`--db-include=`*`имя_бд`*

Задаёт имя базы данных, которая должна восстанавливаться. Все остальные базы данных восстанавливаться не будут, за исключением `template0` и `template1`. Этот параметр можно задать несколько раз, таким образом выбрав для восстановления несколько баз данных.

Репликационные параметры

В этом разделе описываются параметры, относящиеся к резервному копированию с ведомого сервера.

> **Примечание**
>
> Начиная с версии pg_probackup 2.0.24, резервное копирование возможно напрямую с ведомого сервера, без подключения к ведущему, так что эти параметры более не требуются. В предыдущих версиях программа pg_probackup должна была подключаться к ведущему, чтобы определить время восстановления — самый ранний момент, на который можно восстановить согласованное состояние кластера баз данных.

`--master-db=`*`имя_бд`*

Устаревший параметр. Указывает имя базы данных на главном сервере, к которой будет выполнено подключение. Это подключение используется только для управления резервным копированием, так что возможно подключение к любой существующей базе данных. Его можно задать в `pg_probackup.conf` с помощью команды set-config.

По умолчанию: `postgres`, принятое по умолчанию в Postgres Pro имя базы данных

`--master-host=`*`сервер`*

Устаревший параметр. Указывает имя компьютера, на котором работает ведущий сервер.

`--master-port=`*`порт`*

Устаревший параметр. Задаёт TCP-порт или расширение файла Unix-сокета, через который сервер принимает подключения.

По умолчанию: `5432`, стандартный порт Postgres Pro

`--master-user=`*`имя_пользователя`*

Устаревший параметр. Задаёт имя пользователя для подключения.

Если не задано, подразумевается `postgres`, имя пользователя по умолчанию в Postgres Pro

`--replica-timeout=`*`тайм-аут`*

Устаревший параметр. Время ожидания передачи сегментов средствами репликации (в секундах). По умолчанию pg_probackup ожидает 300 секунд. Вы также можете определить этот параметр в файле конфигурации `pg_probackup.conf` с помощью команды set-config.

По умолчанию: `300 sec`

Практические примеры

Во всех последующих примерах предполагается использование удалённого режима по протоколу SSH. Если вы хотите выполнять резервное копирование и восстановление локально, пропустите этап «Настройка SSH-подключения без пароля» и не указывайте никакие параметры `--remote-*`.

Показанные здесь действия выполнялись в среде с Ubuntu 18.04, Postgres Pro 11 и pg_probackup 2.2.0.

- `backup` — роль в Postgres Pro, используемая для подключения к кластеру Postgres Pro.
- `backupdb` — база данных, через которую выполняется подключение к кластеру Postgres Pro.
- `backup_host` — система, где находится каталог резервных копий.
- `backupman` — пользователь в системе `backup_host`, от имени которого выполняются все операции pg_probackup.
- `/mnt/backups` — путь к каталогу резервных копий в системе `backup_host`.
- `postgres_host` — система, в которой работает Postgres Pro.
- `postgres` — пользователь в системе `postgres_host`, от имени которого запускается кластер Postgres Pro.
- `/var/lib/postgresql/11/main` — путь к каталогу данных Postgres Pro в системе `postgres_host`.

Минимальная настройка

Данный сценарий иллюстрирует настройку для выполнения полного и разностного копирования.

1. **Настройте подключение через SSH с** `backup_host` **к** `postgres_host`**:**

```
[backupman@backup_host] ssh-copy-id postgres@postgres_host
```

2. **Настройте кластер Postgres Pro.**

   Из соображений безопасности для выполнения резервного копирования рекомендуется использовать отдельную базу данных.

```
postgres=#
CREATE DATABASE backupdb;
```

   Подключитесь к базе `backupdb`, создайте роль `probackup` и дайте ей следующие права:

```
backupdb=#
BEGIN;
CREATE ROLE backup WITH LOGIN REPLICATION;
GRANT USAGE ON SCHEMA pg_catalog TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.current_setting(text) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.set_config(text, text, boolean) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_is_in_recovery() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_start_backup(text, boolean, boolean) TO
 backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_stop_backup(boolean, boolean) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_create_restore_point(text) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_switch_wal() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_last_wal_replay_lsn() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_current() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_current_snapshot() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_snapshot_xmax(txid_snapshot) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_control_checkpoint() TO backup;
COMMIT;
```

3. **Проинициализируйте каталог резервных копий:**

```
[backupman@backup_host]$ pg_probackup-11 init -B /mnt/backups
INFO: Backup catalog '/mnt/backups' successfully inited
```

4. **Добавьте экземпляр** `pg-11` **в каталог резервных копий:**

```
[backupman@backup_host]$ pg_probackup-11 add-instance -B /mnt/backups --
instance pg-11 --remote-host=postgres_host --remote-user=postgres -D /var/lib/
postgresql/11/main
INFO: Instance 'node' successfully inited
```

5. **Сделайте полную резервную копию:**

```
[backupman@backup_host] pg_probackup-11 backup -B /mnt/backups --instance pg-11
 -b FULL --stream --remote-host=postgres_host --remote-user=postgres -U backup -d
 backupdb
INFO: Backup start, pg_probackup version: 2.2.0, instance: node, backup ID: PZ7YK2,
 backup mode: FULL, wal mode: STREAM, remote: true, compress-algorithm: none,
 compress-level: 1
INFO: Start transferring data files
INFO: Data files are transferred
INFO: wait for pg_stop_backup()
INFO: pg_stop backup() successfully executed
INFO: Validating backup PZ7YK2
INFO: Backup PZ7YK2 data files are valid
INFO: Backup PZ7YK2 resident size: 196MB
INFO: Backup PZ7YK2 completed
```

6. **Взгляните на содержимое каталога копий:**

```
[backupman@backup_host] pg_probackup-11 show -B /mnt/backups --instance pg-11

BACKUP INSTANCE 'pg-11'
```

```
==========================================================================================
 Instance  Version  ID      Recovery Time           Mode   WAL Mode  TLI  Time
 Data   WAL  Zratio  Start LSN  Stop LSN   Status
==========================================================================================
 node      11       PZ7YK2  2019-10-11 19:45:45+03  FULL  STREAM    1/0   11s
 180MB  16MB    1.00  0/3C000028  0/3C000198  OK
```

7. **Сделайте инкрементальную резервную копию в режиме DELTA:**

```
[backupman@backup_host] pg_probackup-11 backup -B /mnt/backups --instance pg-11 -
b delta --stream --remote-host=postgres_host --remote-user=postgres -U backup -d
 backupdb
INFO: Backup start, pg_probackup version: 2.2.0, instance: node, backup ID: PZ7YMP,
 backup mode: DELTA, wal mode: STREAM, remote: true, compress-algorithm: none,
 compress-level: 1
INFO: Parent backup: PZ7YK2
INFO: Start transferring data files
INFO: Data files are transferred
INFO: wait for pg_stop_backup()
INFO: pg_stop backup() successfully executed
INFO: Validating backup PZ7YMP
INFO: Backup PZ7YMP data files are valid
INFO: Backup PZ7YMP resident size: 32MB
INFO: Backup PZ7YMP completed
```

8. **Добавьте несколько параметров в файл конфигурации pg_probackup, чтобы их не надо было задавать в командной строке:**

```
[backupman@backup_host] pg_probackup-11 set-config -B /mnt/backups --instance pg-11
 --remote-host=postgres_host --remote-user=postgres -U backup -d backupdb
```

9. **Сделайте ещё одну инкрементальную копию в режиме DELTA, опустив некоторые из предыдущих параметров:**

```
[backupman@backup_host] pg_probackup-11 backup -B /mnt/backups --instance pg-11 -b
 delta --stream
INFO: Backup start, pg_probackup version: 2.2.0, instance: node, backup ID: PZ7YR5,
 backup mode: DELTA, wal mode: STREAM, remote: true, compress-algorithm: none,
 compress-level: 1
INFO: Parent backup: PZ7YMP
INFO: Start transferring data files
INFO: Data files are transferred
INFO: wait for pg_stop_backup()
INFO: pg_stop backup() successfully executed
INFO: Validating backup PZ7YR5
INFO: Backup PZ7YR5 data files are valid
INFO: Backup PZ7YR5 resident size: 32MB
INFO: Backup PZ7YR5 completed
```

10. **Взгляните на конфигурацию экземпляра:**

```
[backupman@backup_host] pg_probackup-11 show-config -B /mnt/backups --instance
 pg-11

# Backup instance information
pgdata = /var/lib/postgresql/11/main
system-identifier = 6746586934060931492
xlog-seg-size = 16777216
# Connection parameters
pgdatabase = backupdb
pghost = postgres_host
pguser = backup
```

```
# Replica parameters
replica-timeout = 5min
# Archive parameters
archive-timeout = 5min
# Logging parameters
log-level-console = INFO
log-level-file = OFF
log-format-console = PLAIN
log-format-file = PLAIN
log-filename = pg_probackup.log
log-rotation-size = 0
log-rotation-age = 0
# Retention parameters
retention-redundancy = 0
retention-window = 0
wal-depth = 0
# Compression parameters
compress-algorithm = none
compress-level = 1
# Remote access parameters
remote-proto = ssh
remote-host = postgres_host
```

Заметьте, что параметры, не переопределённые командой `set-config`, имеют значения по умолчанию.

11. **Взгляните на содержимое каталога копий:**

```
[backupman@backup_host] pg_probackup-11 show -B /mnt/backups --instance pg-11

========================================================================================
 Instance  Version  ID      Recovery Time           Mode   WAL Mode  TLI  Time
 Data   WAL  Zratio  Start LSN   Stop LSN    Status
========================================================================================
 node      11       PZ7YR5  2019-10-11 19:49:56+03  DELTA  STREAM    1/1   10s
 112kB  32MB    1.00  0/41000028  0/41000160  OK
 node      11       PZ7YMP  2019-10-11 19:47:16+03  DELTA  STREAM    1/1   10s
 376kB  32MB    1.00  0/3E000028  0/3F0000B8  OK
 node      11       PZ7YK2  2019-10-11 19:45:45+03  FULL   STREAM    1/0   11s
 180MB  16MB    1.00  0/3C000028  0/3C000198  OK
```

Версионирование

При разработке pg_probackup используется *семантическое* версионирование.

Авторы

Postgres Professional, Москва, Россия.

Благодарности

Программа pg_probackup основана на pg_arman, которая изначально была написана в NTT, а затем её развивал и поддерживал Микаэль Пакье.

pg_repack

pg_repack — утилита и расширение Postgres Pro Standard для реорганизации таблиц

Синтаксис

`pg_repack` [*параметр*...] [*имя_бд*]

Описание

Модуль *pg_repack* — это расширение Postgres Pro Standard, которое позволяет ликвидировать пустоты в таблицах и индексах и может дополнительно восстанавливать физический порядок кластеризованных индексов. В отличие от CLUSTER и `VACUUM FULL`, оно выполняет эти операции «на ходу», обходясь без исключительных блокировок таблиц в ходе их обработки. К тому же pg_repack действует эффективно, демонстрируя производительность, сравнимую с непосредственным использованием `CLUSTER`.

pg_repack — это альтернативная ветвь развития проекта *https://github.com/reorg/pg_reorg*.

Вы можете выбрать один из следующих способов реорганизации данных:

- Неблокирующая кластеризация (`CLUSTER`) (с упорядочиванием по кластеризующему индексу)
- Упорядочивание по указанным столбцам
- Неблокирующая полная очистка (`VACUUM FULL`) (только упаковка строк)
- Перестроение или перемещение только индексов таблицы

> **Примечание**
>
> Использовать эту утилиту могут только суперпользователи.

> **Примечание**
>
> В целевой таблице должен быть первичный ключ (PRIMARY KEY) или как минимум уникальный индекс по столбцу NOT NULL.

Установка

Для систем Linux pg_repack поставляется с Postgres Pro в виде отдельного пакета, который требует, чтобы был установлен пакет `postgrespro-std-10-server` со всеми зависимостями. Список предоставляемых пакетов и подробное описание установки можно найти в Главе 16. В системах Windows pg_repack устанавливается автоматически в составе Postgres Pro.

Установив pg_repack, загрузите расширение `pg_repack` в базу данных, которую вы хотите обрабатывать, следующим образом:

```
$ psql -c "CREATE EXTENSION pg_repack" -d ваша_база
```

Впоследствии вы можете удалить `pg_repack` из инсталляции Postgres Pro, выполнив `DROP EXTENSION pg_repack`.

Чтобы обновить ранее установленную версию pg_repack>, просто удалите из базы данных старую версию, как описано выше, и установите новую.

Параметры

Параметры реорганизации

`-a`
`--all`

Попытаться перепаковать все базы данных в кластере. Базы данных, в которых расширение `pg_repack` не установлено, будут пропущены.

`-t` *таблица*
`--table=`*таблица*

Реорганизовать только указанную таблицу (таблицы). Реорганизовать несколько таблиц можно, добавив несколько ключей `-t`. По умолчанию реорганизуются все подходящие таблицы в целевых базах данных.

`-c` *схема*
`--schema=`*схема*

Перепаковать таблицы только в указанной схеме (схемах). Перепаковать несколько схем можно, добавив несколько ключей `-c`. Можно использовать в сочетании с `--tablespace` для перемещения таблиц в другое табличное пространство.

`-o` *столбец*`[, ...]`
`--order-by=`*столбец*`[, ...]`

Выполнить неблокирующую кластеризацию (`CLUSTER`), упорядочивая данные по указанным столбцам.

`-n`
`--no-order`

Выполнить неблокирующую полную очистку (`VACUUM FULL`). Начиная с версии 1.2, это производится по умолчанию для некластеризованных таблиц.

`-N`
`--dry-run`

Только показать, какие таблицы будут перепакованы, и завершиться.

`-j` *num_jobs*
`--jobs=`*num_jobs*

Установить заданное количество дополнительных соединений к Postgres Pro Standard и использовать эти дополнительные подключения для перестраивания индексов таблиц в параллельном режиме. Параллельное перестроение индексов поддерживается только при реорганизации таблиц полностью, без ключей `--index` или `--only-indexes`. Если ваш сервер имеет несколько процессорных ядер и быструю дисковую подсистему, параллельный режим может быть полезен для ускорения pg_repack.

`-s` *табличное_пространство*
`--tablespace=`*табличное_пространство*

Перенести перепакованные таблицы в заданное табличное пространство: по сути это неблокирующая версия команды `ALTER TABLE ... SET TABLESPACE`. Индексы таблиц остаются в исходном табличном пространстве, если только дополнительно не указан ключ `--moveidx`.

`-S`
`--moveidx`

Также перенести индексы перепакованных таблиц в табличное пространство, заданное ключом `--tablespace`.

`-i` *индекс*
`--index=`*индекс*

Перепаковать только указанный индекс. Перепаковать несколько индексов можно, добавив несколько ключей `-i`. Можно использовать в сочетании с `--tablespace` для перемещения индексов в другое табличное пространство.

`-x`
`--only-indexes`

Перепаковать только индексы таблиц, заданных ключами `--table`.

`-T` *сек*
`--wait-timeout=`*сек*

Расширению pg_repack необходимо получить исключительную блокировку в конце реорганизации. Этот параметр определяет, сколько секунд pg_repack будет ждать получения этой блокировки. Если за это время блокировка не будет получена, pg_repack принудительно отменит конфликтующие запросы. Если же вы используете Postgres Pro или PostgreSQL версии 8.4 и новее, pg_repack прибегнет к вызову `pg_terminate_backend()`, чтобы отключить все оставшиеся фоновые процессы, после того, как это время истечёт дважды. Значение по умолчанию: 60 сек.

`-Z`
`--no-analyze`

Не выполнять `ANALYZE` после полной реорганизации таблицы. В отсутствие этого ключа после реорганизации `ANALYZE` выполняется.

Параметры подключения

`[-d]` *имя_бд*
`[--dbname=]`*имя_бд*

Указывает имя базы данных для реорганизации. Если оно не указано, и параметр `-a` (или `--all`) не используется, то имя базы берётся из переменной окружения `PGDATABASE`. Если и эта переменная не задана, выбирается имя подключающегося пользователя.

`-h` *сервер*
`--host=`*сервер*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *порт*
`--port=`*порт*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

`-U` *имя_пользователя*
`--username=`*имя_пользователя*

Имя пользователя для подключения.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как pg_repack запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, pg_repack лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

Общие параметры

`-e`
`--echo`

Выводить команды, которые pg_repack формирует и передаёт на сервер.

`-E` *уровень*
`--elevel=`*уровень*

Устанавливает уровень выводимых сообщений: `DEBUG`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL` и `PANIC`. Уровень по умолчанию: `INFO`.

`--help`

Вывести справку об аргументах командной строки pg_repack и завершиться.

`-V`
`--version`

Вывести версию pg_repack и завершиться.

Переменные окружения

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 31.14).

Примеры

Выполнение неблокирующей кластеризации (`CLUSTER`) всех кластеризованных таблиц и полной очистки (`VACUUM FULL`) всех некластеризованных таблиц в базе данных `test`:

```
$ pg_repack test
```

Выполнение неблокирующей полной очистки (`VACUUM FULL`) в таблицах `foo` и `bar` в базе данных `test` (кластерные индексы, если они есть, игнорируются):

```
$ pg_repack --no-order --table foo --table bar test
```

Перемещение всех индексов таблицы `foo` в табличное пространство `tbs`:

```
$ pg_repack -d test --table foo --only-indexes --tablespace tbs
```

Перемещение указанного индекса в табличное пространство `tbs`:

```
$ pg_repack -d test --index idx --tablespace tbs
```

Диагностика

В случае ошибок pg_repack выдаёт соответствующие сообщения. В следующем списке поясняются причины ряда ошибок.

В случае неисправимых ошибок вам потребуется провести очистку вручную. Для этого просто удалите pg_repack из базы данных, а затем снова установите это расширение.

Для Postgres Pro или PostgreSQL версии 9.1 и новее выполните:

```
DROP EXTENSION pg_repack CASCADE
```

в базе данных, где произошла ошибка. Затем выполните:

```
CREATE EXTENSION pg_repack
```

Для предыдущих версий загрузите скрипт $SHAREDIR/contrib/uninstall_pg_repack.sql в базу данных, где произошла ошибка, а затем повторно загрузите $SHAREDIR/contrib/pg_repack.sql.

```
INFO: database "db" skipped:
pg_repack VER is not installed in the database:
pg_repack is not installed in the database when the --all option is specified.
```

Создайте расширение pg_repack в базе данных.

```
ERROR: pg_repack VER is not installed in the database:
pg_repack is not installed in the database specified by --dbname
```

Создайте расширение pg_repack в базе данных.

```
ERROR: program 'pg_repack V1' does not match database library 'pg_repack V2':
There is a mismatch between the pg_repack binary and the database library
(.so or .dll).
```

Обнаруженное несоответствие версий может быть связано с нахождением некорректной программы в переменной PATH или обращением к неправильной базе данных. Проверьте корректность каталога программы и базы данных; если всё верно, возможно, вам придётся переустановить pg_repack.

```
ERROR: extension 'pg_repack V1' required, found 'pg_repack V2':
The SQL extension found in the database does not match the version required by the
 pg_repack program.
```

Столкнувшись с такой ошибкой, удалите расширение из базы данных и пересоздайте его, как описано в Разделе «Установка».

```
ERROR: relation "table" must have a primary key or not-null unique keys:
The target table doesn't have a PRIMARY KEY or any UNIQUE constraints defined.
```

Создайте в указанной таблице ограничение PRIMARY KEY или UNIQUE.

```
ERROR: query failed: ERROR: column "col" does not exist:
The target table doesn't have columns specified by --order-by option.
```

Укажите существующие столбцы.

```
WARNING: the table "tbl" already has a trigger called z_repack_trigger:
```

```
The trigger was probably installed during a previous attempt
to run pg_repack on the table which was interrupted
and for some reason failed to clean up the temporary objects.
```

Вы можете ликвидировать все временные объекты, удалив и создав расширение заново: это описано подробно в Разделе «Установка».

```
WARNING: trigger "trg" conflicting on table "tbl":
The target table has a trigger whose name follows z_repack_trigger
in alphabetical order.
```

Триггер `z_repack_trigger` должен быть последним триггером в наборе BEFORE. Переименуйте свой триггер, чтобы он шёл в алфавитном порядке перед `pg_repack`; это можно сделать командой:

```
ALTER TRIGGER zzz_my_trigger
 ON таблица RENAME TO yyy_my_trigger;
```

```
ERROR: Another pg_repack command may be running on the table. Please try again later.
```

Когда две команды `pg_repack` выполняются одновременно, возможна взаимоблокировка. В таком случае стоит попробовать выполнять эти команды с некоторой задержкой.

```
WARNING: Cannot create index "schema"."index_xxxxx", already exists
DETAIL: An invalid index may have been left behind by a previous
pg_repack on the table which was interrupted. Please use DROP INDEX
"schema"."index_xxxxx" to remove this index and try again.
```

Похоже, что в базе данных остался временный индекс, созданный программой pg_repack, но он не удаляется автоматически во избежание недоразумений. Если этот индекс действительно был создан старым заданием pg_repack и по какой-то причине не удалился, вы можете просто выполнить `DROP INDEX` и повторить команду `repack` ещё раз.

Ограничения

pg_repack имеет следующие ограничения.

Временные таблицы

pg_repack не может реорганизовывать временные таблицы.

Индексы GiST

pg_repack не может кластеризовать таблицу согласно индексу `GiST`.

Команды DDL

Пока работает pg_repack, нельзя выполнять команды DDL с целевыми таблицами, *за исключением* `VACUUM` и `ANALYZE`. Для реализации этого ограничения pg_repack устанавливает блокировку `ACCESS SHARE` в целевой таблице на время реорганизации таблицы.

Если вы используете версию расширения 1.1.8 или старее, воздержитесь от выполнения каких-либо команд DDL с целевыми таблицами в процессе работы pg_repack. Во многих случаях pg_repack выдаст ошибку и отменит свои изменения корректно, но с этими старыми версиями в некоторых случаях было возможно разрушение данных.

См. также

clusterdb, vacuumdb

vacuumlo

vacuumlo — удалить потерянные большие объекты из базы данных Postgres Pro

Синтаксис

`vacuumlo` [*параметр*...] *имя_бд*...

Описание

Программа vacuumlo представляет собой простую утилиту, которая удаляет все «потерянные» большие объекты из базы данных Postgres Pro. Потерянным большим объектом (БО) считается такой БО, OID которого не фигурирует ни в каком столбце `oid` или `lo` в базе данных.

Если вы применяете эту утилиту, вас также может заинтересовать триггер `lo_manage` в модуле lo. Триггер `lo_manage` полезен тем, что стремится предотвратить образование потерянных БО.

Обработке подвергаются все базы данных, перечисленные в командной строке.

Параметры

vacuumlo принимает следующие аргументы командной строки:

`-l` *предел*

Удалять в одной транзакции ограниченное количество больших объектов (максимальное количество задаёт *предел*, по умолчанию 1000). Так как сервер запрашивает блокировку для каждого удаляемого БО, удаление слишком большого количества БО в одной транзакции чревато превышением лимита max_locks_per_transaction. Если вы всё же хотите, чтобы все удаления происходили в одной транзакции, установите этот предел, равным нулю.

`-n`

Не удалять ничего, только показать, какие операции должны были выполняться.

`-v`

Выводить подробные сообщения о прогрессе.

`-V`
`--version`

Вывести версию vacuumlo и завершиться.

`-?`
`--help`

Вывести справку об аргументах командной строки vacuumlo и завершиться.

vacuumlo также принимает в командной строке следующие аргументы, задающие параметры подключения:

`-h` *компьютер*

Адрес сервера баз данных.

`-p` *порт*

Порт сервера баз данных.

`-U` *имя_пользователя*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как vacuumlo запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, vacuumlo лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

Замечания

Программа vacuumlo работает следующим образом: сначала vacuumlo строит временную таблицу, содержащую все OID больших объектов в выбранной базе данных. Затем она сканирует все столбцы в базе данных, имеющие тип `oid` или `lo`, и удаляет соответствующие записи из временной таблицы. (Замечание: рассматриваются только типы именно с такими именами, а не, например, домены на их базе.) Оставшиеся записи во временной таблице указывают на потерянные БО, которые затем и удаляются.

Автор

Питер Маунт `<peter@retep.org.uk>`

G.2. Серверные приложения

В этом разделе рассматриваются приложения, связанные с функциональностью сервера Postgres Pro Standard. Они обычно запускаются в той же системе, где работает сервер баз данных. Обратитесь также к Справке: «Серверные приложения Postgres Pro» за информацией о серверных приложениях, относящихся к ядру дистрибутива Postgres Pro Standard.

mamonsu

mamonsu — агент мониторинга для сбора метрик операционной системы и Postgres Pro.

Синтаксис

`mamonsu agent` [*действие_агента*]

`mamonsu bootstrap -M` *пользователь_mamonsu* [`-c | --config`] [`-x | --create-extensions`] [*параметры_подключения*]

`mamonsu export {template | config}` *имя_файла* [*параметры_экспорта*]

`mamonsu report` [*параметры_отчёта*] [*параметры_подключения*]

`mamonsu tune` [*параметры_оптимизации*] [*параметры_подключения*]

`mamonsu upload` [*параметры_передачи*]

`mamonsu zabbix dashboard upload` *имя_шаблона*

`mamonsu zabbix {template | host | hostgroup}` *действие_сервера*

`mamonsu zabbix item {error | lastvalue | lastclock}` *ид_узла*

`mamonsu zabbix version`

`mamonsu --version`

`mamonsu --help`

Описание

Приложение mamonsu представляет собой агент мониторинга, собирающий метрики Postgres Pro и системы, которые можно визуализировать на сервере Zabbix. В отличие от стандартного Zabbix Agent, который тоже можно настроить для сбора метрик Postgres Pro, mamonsu использует одно подключение к базе данных, что позволяет минимизировать влияние на целевую систему.

Агент mamonsu включает следующие компоненты:

- Руководящий процесс, наблюдающий за активностью системы и базы данных.
- Модули, определяющие, какие метрики Postgres Pro и системы будут собираться.
- Файлы конфигурации и шаблона, в которых определяется, какие модули будут использоваться и как будут представляться собранные данные.

mamonsu является активным агентом, что означает, что он передаёт серверу Zabbix данные сразу, по мере их сбора. В mamonsu имеется набор готовых метрик для мониторинга Postgres Pro и системы, но вы можете разработать и дополнительные модули для сбора других интересующих вас метрик.

В mamonsu также реализован интерфейс командной строки для управления некоторыми параметрами сервера Zabbix, для получения общего представления о конфигурации целевой системы, а также для оптимизации параметров Postgres Pro и системы «на лету». Список всех команд mamonsu приведён в Разделе «Справка по командной строке».

Установка и начальная настройка

> **Важно**
>
> Перед началом установки рекомендуется внимательно ознакомиться с Замечаниями о совместимости.

Для использования mamonsu вы должны создать учётную запись Zabbix и настроить сервер Zabbix, как описано в *документации Zabbix*. Естественно, если вы планируете наблюдать за метриками Postgres Pro, у вас должен работать экземпляр Postgres Pro. Если вам нужны инструкции по установке и настройке кластера Postgres Pro, обратитесь к Главе 16 и Разделу 17.2.

> **Примечание**
>
> Тогда как метрики Postgres Pro mamonsu может собирать с удалённого кластера, метрики системы собираются только локально. Поэтому если вы решите собирать метрики удалённого экземпляра Postgres Pro, во избежание путаницы обязательно отключите системные метрики, иначе оба набора будут представлены в Zabbix как метрики одного узла.

Предварительно собранный пакет mamonsu распространяется в составе Postgres Pro Standard, но устанавливается отдельным инсталлятором. Завершив установку mamonsu, выполните следующие действия, чтобы начать сбор метрик:

1. **По желанию произведите специальную подготовку mamonsu**

   Если вы опустите этот этап, метрики можно будет собирать только с правами суперпользователя, что не рекомендуется.

   На этом этапе команда `bootstrap` задаёт пользователя mamonsu, создаёт схему `mamonsu` в базе данных, заданной явно или по умолчанию, а также расширения и объекты, необходимые для мониторинга.

   Команда `bootstrap` не имеет обязательных параметров. При запуске без параметров она берёт необходимые значения из файла конфигурации (по умолчанию — `/etc/mamonsu/agent.conf`). Ниже приведён простой пример подготовительной процедуры:

   1. В разделе `[postgres]` файла `agent.conf` укажите параметры подключения для кластера Postgres Pro, за которым вы собираетесь наблюдать. За дополнительными сведениями обратитесь к разделу Параметры подключения.
   2. Выполните следующую команду для подготовки mamonsu:

      ```
      mamonsu bootstrap
      ```

      Эта команда создаст пользователя `mamonsu` с паролем `mamonsu`, а также функции мониторинга в базе данных, указанной в `agent.conf`, и предоставит право выполнять их пользователю `mamonsu`.

   В следующем примере параметры `bootstrap` передаются явно.

   1. Создайте непривилегированного пользователя базы данных для mamonsu. Например:

      ```
      CREATE USER mamonsu_user WITH PASSWORD 'mamonsu_password';
      ```

   2. Создайте базу данных, которая будет применяться для установления подключения к Postgres Pro. Например:

      ```
      CREATE DATABASE mamonsu_database OWNER mamonsu_user;
      ```

   3. Выполните следующую команду для подготовки mamonsu:

      ```
      mamonsu bootstrap -M mamonsu_user -x -c /etc/mamonsu/mamonsu.conf -d
       mamonsu_database -U postgres --host myhost --port 5432
      ```

      Для получения подробной информации о параметрах `-M`, `-x` и `-c` см. `bootstrap`. За дополнительными сведениями о других параметрах обратитесь к разделу Параметры подключения. Эта команда создаст функции мониторинга в *`базе_mamonsu`* и предоставит право выполнять их *`пользователю_mamonsu`*. Кроме того, будет создано расширение pg_buffercache, позволяющее собирать метрики общего кеша буферов. При этом задаётся другой путь к файлу конфигурации.

В результате суперпользователь не требуется для дальнейших подключений. Обратите внимание, что в этой базе данных создаётся несколько таблиц в схеме `mamonsu`. Не удаляйте эти таблицы, так как они необходимы для работы mamonsu.

2. **Настройте mamonsu**

   Отредактируйте конфигурационный файл `agent.conf`, по умолчанию размещённый в каталоге `/etc/mamonsu/`.

   - Настройте параметры, связанные с Zabbix. Значение `address` должно указывать на работающий сервер Zabbix, а в поле `client` должно задаваться имя узла Zabbix. Узлы, доступные для вашей учётной записи Zabbix, вы можете увидеть в его веб-интерфейсе на странице **Configuration** (Настройка) **> Hosts** (Узлы сети).

     ```
     [zabbix]
     ; включён по умолчанию
     enabled = True
     client = имя_узла_zabbix
     address = сервер_zabbix
     ```

   - По умолчанию mamonsu будет собирать и метрики Postgres Pro, и системные метрики. При необходимости вы можете запретить сбор метрик первого или второго типа, установив в файле `agent.conf` для параметра `enabled` значение `False` в разделе `[postgres]` или `[system]`, соответственно.

     ```
     [system]
     ; включён по умолчанию
     enabled = True
     ```

     > **Примечание**
     >
     > Тогда как метрики Postgres Pro mamonsu может собирать с удалённого кластера, метрики системы собираются только локально. Поэтому если вы решите собирать метрики удалённого экземпляра Postgres Pro, во избежание путаницы обязательно отключите системные метрики, иначе оба набора будут представлены в Zabbix как метрики одного узла.

   - Если вы будете собирать метрики Postgres Pro, необходимо задать параметры подключения к кластеру Postgres Pro, за которым вы намерены наблюдать. Для этого задайте в полях `user`, `password` и `database` соответственно имя пользователя (*`mamonsu_user`*), пароль (*`mamonsu_password`*) и базу данных (*`mamonsu_database`*), указанные при подготовке. Если вы пропустили подготовительную процедуру, задайте имя и пароль суперпользователя, а также целевую базу.

     ```
     [postgres]
     ; включён по умолчанию
     enabled = True
     user = пользователь_mamonsu
     database = база_mamonsu
     password = пароль_mamonsu
     port = 5432
     ```

   Это базовые параметры, позволяющие начать использование mamonsu. Кроме них можно настроить и другие параметры mamonsu, описанные в Разделе «Параметры конфигурации».

3. **Настройте представление метрик на сервере Zabbix**

   1. Создайте шаблон, определяющий, как будут представляться собранные метрики на сервере Zabbix:

```
mamonsu export template template.xml
```

mamonsu формирует файл шаблона `template.xml` в текущем каталоге. По умолчанию отображаемым в Zabbix именем шаблона будет `PostgresPro-`*`OS`*, где *`OS`* — название вашей операционной системы. Чтобы получить шаблон с другим отображаемым именем, в показанную выше команду можно добавить аргумент `--template-name`.

2. При желании укажите учётные данные в Zabbix в следующих переменных окружения в той системе, где настраивается мониторинг:

   - Задайте в переменных `ZABBIX_USER` и `ZABBIX_PASSWD` имя и пароль вашей учётной записи Zabbix, соответственно.
   - Задайте в переменной `ZABBIX_URL` адрес `http://zabbix/`

   Если этого не сделать, вам нужно будет указывать следующие параметры при каждом вызове любой подкоманды `mamonsu zabbix`:

   ```
   --url=http://zabbix/ --user=имя_пользователя_zabbix --
   password=пароль_пользователя_zabbix
   ```

3. Загрузите шаблон `template.xml` на сервер Zabbix.

   ```
   mamonsu zabbix template export template.xml
   ```

   Шаблон также можно загрузить через веб-интерфейс Zabbix: воспользуйтесь своей учётной записью в Zabbix и перейдите на страницу **Templates** (Шаблоны) > **Import** (Импорт).

4. Привяжите полученный шаблон к узлу, за которым будете наблюдать.

   Для этого в веб-интерфейсе Zabbix выберите нужный узел, перейдите на страницу **Templates** (Шаблоны) > **Add** (Добавить), выберите шаблон и затем щёлкните **Update** (Обновить).

> **Подсказка**
>
> Если вы хотите связать шаблон с создаваемым узлом в Zabbix, вы можете сделать это в командной строке, используя команды `mamonsu zabbix`. Подробнее это описывается в Подразделе «Управление параметрами сервера Zabbix в командной строке».

Завершив настройку, запустите mamonsu. Например, в системах Linux вы можете запустить mamonsu в виде службы так:

```
service mamonsu start
```

mamonsu считывает все параметры из файла конфигурации mamonsu и начинает мониторинг системы.

Справка по командной строке

agent

Синтаксис:

```
mamonsu agent { metric-list | metric-get имя_метрики | version }
```

Предоставляет информацию о собираемых метриках из командной строки. Вы можете задать одно из следующих действий:

`metric-list`

Вывести список метрик, которые собирает mamonsu. Вывод этой команды включает ключ метрики, её последнее значение и время получения этого значения.

`metric-get` *`имя_метрики`*

Получить последнее значение указанной метрики. Список доступных метрик можно получить с помощью команды `metric-list`.

`version`

Вывести версию mamonsu.

bootstrap

Синтаксис:

```
mamonsu bootstrap -M имя пользователя
        [-c[ файл конфигурации] | --config[=файл конфигурации]]
        [-x | --create-extensions] [параметры подключения]
```

Подготовить mamonsu к работе. Данная команда может принимать следующие параметры:

`-M`

Задаёт имя пользователя, который будет владеть всеми процессами mamonsu. Указанный пользователь не должен быть привилегированным.

`-c` *`файл_конфигурации`*
`--config=`*`файл_конфигурации`*

Задаёт файл `agent.conf`, в котором раздел `[postgres]` содержит имя базы данных, которое будет использоваться, если параметр `--dbname` не указан среди *`параметров подключения`*.

Значение по умолчанию: `/etc/mamonsu/agent.conf`

`-x`
`--create-extensions`

Создаёт вспомогательные расширения. Будет создано расширение pg_buffercache, которое больше не создаётся по умолчанию. Для Postgres Pro версии 12 или выше команда `bootstrap -x` создаст функции для работы с pgpro_stats и предоставит права на их выполнение пользователю `mamonsu`.

`параметры_подключения`

Необязательные параметры подключения, задаваемые в командной строке.

export

Синтаксис:

```
mamonsu export config имя_файла.conf  [--add-plugins=каталог_модулей]
mamonsu export template имя_файла.xml [--add-plugins=каталог_модулей]
                                      [--template-name=имя_шаблона]
                                      [--application=имя_приложения]
                                      [--old-zabbix]
```

Сгенерировать файл конфигурации или файл шаблона для набора метрик. Для настройки набора метрик можно передать дополнительные параметры:

`--add-plugins=`*`каталог_модулей`*

Собирать метрики, определённые в дополнительных модулях, расположенных в заданном *`каталоге_модулей`*. Если вы будете использовать какие-либо дополнительные модули, этот параметр нужно передавать при формировании файла конфигурации и шаблона.

`--template-name=`*`имя_шаблона`*

Задаёт название шаблона, которое будет отображаться на сервере Zabbix.

Значение по умолчанию: `PostgresPro-`*`OS`*, где *`OS`* — название вашей операционной системы

`--application=`*`имя_приложения`*

Задаёт идентификатор, с которым собранные метрики будут отображаться на сервере Zabbix.

Значение по умолчанию: `App-PostgresPro-`*`OS`*, где *`OS`* — название вашей операционной системы

`--old-zabbix`

Экспортировать шаблон для сервера Zabbix версии 4.2 или старее. По умолчанию шаблон генерируется в формате, совместимом с Zabbix 4.4 и новее.

export zabbix-parameters

Синтаксис:

```
mamonsu export zabbix-parameters имя_файла.conf [--add-plugins=каталог_модулей]
                                      [--plugin-type={pg | sys | all}]
                                      [--pg-version=версия]
                                      [--config=файл_конфигурации]
```

Экспортировать конфигурацию метрик для использования со стандартным *агентом Zabbix*. Для настройки сбора метрик можно передать дополнительные параметры:

`--add-plugins=`*`каталог_модулей`*

Собирать метрики, определённые в дополнительных модулях, расположенных в заданном *`каталоге_модулей`*. Если вы будете использовать какие-либо дополнительные модули, этот параметр нужно передавать при формировании файла конфигурации и шаблона.

`--plugin-type={pg | sys | all}`

Типы метрик, которые должны собираться:

- `pg` — метрики Postgres Pro.
- `sys` — системные метрики.
- `all` — все метрики.

По умолчанию: `all`

`--pg-version=`*`версия`*

Задаёт основную версию сервера, для которого будет настраиваться набор метрик. Программа `mamonsu` может собирать метрики во всех поддерживаемых версиях Postgres Pro, а также в PostgreSQL, начиная с версии 9.5.

Значение по умолчанию: `10`

`--config=`*`файл_конфигурации`*

Задаёт путь к файлу `agent.conf`, который будет источником определений метрик.

Значение по умолчанию: `/etc/mamonsu/agent.conf`

export zabbix-template

Синтаксис:

```
mamonsu export zabbix-template имя_файла.conf [--add-plugins=каталог_модулей]
                                      [--plugin-type={pg | sys | all}]
                                      [--template-name=имя_шаблона]
                                      [--application=имя_приложения]
                                      [--config=файл_конфигурации]
```

```
[--old-zabbix]
```

Экспортировать шаблон для использования со стандартным агентом Zabbix. Для настройки конфигурации можно передать дополнительные параметры:

`--add-plugins=`*`каталог_модулей`*

Собирать метрики, определённые в дополнительных модулях, расположенных в заданном *`каталоге_модулей`*. Если вы будете использовать какие-либо дополнительные модули, этот параметр нужно передавать при формировании файла конфигурации и шаблона.

`--plugin-type={pg | sys | all}`

Типы метрик, которые должны собираться:

- `pg` — метрики Postgres Pro.
- `sys` — системные метрики.
- `all` — все метрики.

По умолчанию: `all`

`--template-name=`*`имя_шаблона`*

Задаёт название шаблона, которое будет отображаться на сервере Zabbix.

Значение по умолчанию: `PostgresPro-`*`OS`*, где *`OS`* — название вашей операционной системы

`--application=`*`имя_приложения`*

Задаёт идентификатор, с которым собранные метрики будут отображаться на сервере Zabbix.

Значение по умолчанию: `App-PostgresPro-`*`OS`*, где *`OS`* — название вашей операционной системы

`--config=`*`файл_конфигурации`*

Задаёт путь к файлу `agent.conf`, который будет источником определений метрик.

Значение по умолчанию: `/etc/mamonsu/agent.conf`

`--old-zabbix`

Экспортировать шаблон для сервера Zabbix версии 4.2 или старее. По умолчанию шаблон генерируется в формате, совместимом с Zabbix 4.4 и новее.

report

Синтаксис:

```
mamonsu report [--run-system=логическое_значение] [--run-postgres=логическое_значение]
               [--print-report=логическое_значение] [--report-path=файл_отчёта]
               [--disable-sudo] [параметры_подключения]
```

Выдать подробный отчёт об оборудовании, операционной системе, использовании памяти и других характеристиках объекта мониторинга, а также конфигурацию Postgres Pro.

Отчёт настраивается с помощью следующих необязательных параметров:

`--run-system=`*`логическое_значение`*

Включить в генерируемый отчёт информацию о системе.

По умолчанию: `True` (вкл.)

`--run-postgres=`*`логическое_значение`*

Включить в генерируемый отчёт информацию о Postgres Pro.

По умолчанию: `True` (вкл.)

`--print-report=`*логическое_значение*

Вывести отчёт в stdout.

По умолчанию: `True` (вкл.)

`--report-path=`*файл_отчёта*

Вывести отчёт в указанный файл.

Путь по умолчанию: `/tmp/report.txt`

`--disable-sudo`

Собирать при формировании отчёта только те данные, для получения которых не нужны права суперпользователя. Этот режим поддерживается только в системах Linux.

параметры_подключения

Необязательные параметры подключения, задаваемые в командной строке.

tune

Синтаксис:

```
mamonsu tune [--dry-run] [--disable-sudo] [--log-level {INFO|DEBUG|WARN}]
             [--dont-tune-pgbadger] [--dont-reload-postgresql]
             [параметры_подключения]
```

Оптимизировать конфигурацию Postgres Pro и системы, исходя из собранной статистики. Настроить оптимизацию позволяют следующие параметры:

`--dry-run`

Вывести параметры конфигурации системы и Postgres Pro, которые будут настроены, но не изменять их фактически.

`--disable-sudo`

Оптимизировать только те параметры, для изменения которых не нужны права суперпользователя. Этот режим поддерживается только в системах Linux.

`--dont-tune-pgbadger`

Не оптимизировать параметры pgbadger.

`--log-level { INFO | DEBUG | WARN}`

Изменить уровень детализации сообщений.

Значение по умолчанию: `INFO`

`--dont-reload-postgresql`

Не допустить выполнения приложением mamonsu функции `pg_reload_conf()`. В этом случае изменения некоторых параметров, которые вступают в силу только после перезагрузки конфигурации Postgres Pro, не начнут действовать немедленно.

параметры_подключения

Необязательные параметры подключения, задаваемые в командной строке.

upload

Синтаксис:

```
mamonsu upload [--zabbix-file=файл_метрик]
               [--zabbix-address=адрес_zabbix] [--zabbix-port=порт]
               [--zabbix-client=имя_узла_zabbix] [--zabbix-log-level={INFO|DEBUG|WARN}]
```

Передать данные метрик, ранее сохранённые в файле, на сервер Zabbix для визуализации. Более подробно получение файла метрик рассматривается в Подразделе «Параметры вывода».

Эта команда может принимать следующие параметры:

`--zabbix-address=адрес_zabbix`

Адрес сервера Zabbix.

Значение по умолчанию: `localhost`

`--zabbix-port=порт`

Порт сервера Zabbix.

Значение по умолчанию: 10051

`--zabbix-file=файл_метрик`

Имя текстового файла с метриками, которые можно визуализировать, например `localhost.log`.

`--zabbix-client=имя_узла_zabbix`

Имя узла в Zabbix.

Значение по умолчанию: `localhost`

`--zabbix-log-level={INFO|DEBUG|WARN}`

Изменить уровень детализации сообщений.

Значение по умолчанию: `INFO`

zabbix dashboard upload

Синтаксис:

```
mamonsu dashboard upload имя_шаблона
```

Загрузить панель Zabbix с метриками mamonsu в шаблон на сервере Zabbix версии 6.0 или выше.

zabbix item

Синтаксис:

```
mamonsu zabbix item {error | lastvalue | lastclock } имя_узла
```

Просмотреть заданное свойство последних метрик, полученных сервером Zabbix для указанного узла.

zabbix version

Синтаксис:

```
mamonsu zabbix version
```

Получить версию сервера Zabbix, к которой подключён агент mamonsu.

zabbix host

Синтаксис:

```
mamonsu zabbix host list
```

```
mamonsu zabbix host show имя_узла
mamonsu zabbix host id имя_узла
mamonsu zabbix host delete ид_узла
mamonsu zabbix host create имя_узла ид_группы_узлов ид_шаблона адрес_mamonsu
mamonsu zabbix host info {templates | hostgroups | graphs | items} ид_узла
```

Позволяет управлять узлами Zabbix, выполняя одно из действий, описанных в Подразделе «Управление сервером Zabbix».

zabbix hostgroup

Синтаксис:

```
mamonsu zabbix hostgroup list
mamonsu zabbix hostgroup show имя_группы_узлов
mamonsu zabbix hostgroup id имя_группы_узлов
mamonsu zabbix hostgroup delete ид_группы_узлов
mamonsu zabbix hostgroup create имя_группы_узлов
```

Позволяет управлять группами узлов Zabbix, выполняя одно из действий, описанных в Подразделе «Управление сервером Zabbix».

zabbix template

Синтаксис:

```
mamonsu zabbix template list
mamonsu zabbix template show имя_шаблона
mamonsu zabbix template id имя_шаблона
mamonsu zabbix template delete ид_шаблона
mamonsu zabbix template export файл
```

Позволяет управлять шаблонами Zabbix, выполняя одно из действий, описанных в Подразделе «Управление сервером Zabbix».

--version

Синтаксис:

```
mamonsu --version
```

Вывести версию `mamonsu`.

--help

Синтаксис:

```
mamonsu --help
```

Вывести справку по параметрам командной строки `mamonsu`.

Параметры подключения

При вызове вышеуказанных команд *параметры_подключения* в командной строке задают параметры подключения к целевому кластеру Postgres Pro. *параметры_подключения* могут содержать аргументы `--host`, `--port`, `--dbname` (`-d`), `--username` (`-U`), и `--password` (`-W`). В аргументе `--dbname` (`-d`) должна указываться `mamonsu_database`, созданная для наблюдения. Заметьте, что в аргументе `--username` (`-U`) необходимо передать имя суперпользователя в данном кластере.

Если вы не зададите *параметры_подключения*, mamonsu будет использовать соответствующие параметры из файла конфигурации.

Управление сервером Zabbix

Используя mamonsu, вы можете управлять некоторой функциональностью Zabbix из командной строки. А именно, вы можете создавать или удалять узлы и группы узлов Zabbix, а также

генерировать, импортировать и удалять шаблоны Zabbix с помощью описанных ниже команд. В них *имя_объекта* должно соответствовать типу объекта Zabbix в команде: template (шаблон), host (узел) или hostgroup (группа узлов).

`list`

Вывести список доступных шаблонов, узлов или групп узлов.

`show` *имя_объекта*

Вывести информацию об указанном шаблоне, узле или группе узлов.

`id` *имя_объекта*

Показать идентификатор заданного объекта, назначаемый автоматически сервером Zabbix.

`delete` *ид_объекта*

Удалить объект с заданным идентификатором.

`create` *имя_группы_узлов*
`create` *имя_узла ид_группы_узлов ид_шаблона адрес_mamonsu*

Создать новый узел или группу узлов.

`export` *имя_шаблона*

Сформировать шаблон Zabbix.

`info {templates | hostgroups | graphs | items}` *ид_узла*

Вывести подробную информацию о шаблонах, группах узлов, графиках и метриках, связанных с узлом с заданным идентификатором.

Параметры конфигурации

Файл конфигурации `agent.conf` по умолчанию располагается в каталоге `/etc/mamonsu`. Он содержит несколько групп параметров, определяющих, какие метрики собирать и как выводить собранные данные:

- параметры подключения
- параметры вывода
- параметры модулей

> **Важно**
>
> Только пользователь `mamonsu` может читать и редактировать файл `agent.conf`.

Все параметры должны задаваться в формате *параметр = значение*.

Параметры подключения агента

[postgres]

В разделе `[postgres]` настраивается сбор метрик Postgres Pro и могут содержаться следующие параметры:

`enabled`

Включает/отключает сбор метрик Postgres Pro при значении, равном `True` или `False`, соответственно.

По умолчанию: `True` (вкл.)

`user`

Пользователь, от имени которого будут запускаться процессы наблюдения за кластером. Это должен быть пользователь mamonsu, заданный при подготовке, либо суперпользователь, если команда `bootstrap` не запускалась.

Имя по умолчанию: `postgres`

`password`

Пароль указанного пользователя.

`database`

База данных, через подключение к которой будут собираться метрики.

Имя по умолчанию: `postgres`

`host`

Адрес сервера базы данных.

Значение по умолчанию: `localhost`

`port`

Целевой порт на сервере.

Значение по умолчанию: 5432

`application_name`

Имя приложения, идентифицирующее процесс mamonsu, выполняющийся в кластере Postgres Pro.

Значение по умолчанию: `mamonsu`

`query_timeout`

Значение statement_timeout для сеанса mamonsu, в секундах. Если запрос метрики Postgres Pro не успевает выполниться за указанное время, он прерывается.

Значение по умолчанию: 10

[system]

В разделе `[system]` настраивается сбор системных метрик и могут содержаться следующие параметры:

`enabled`

Включает/отключает сбор системных метрик при значении, равном `True` или `False`, соответственно.

По умолчанию: `True` (вкл.)

[zabbix]

В разделе `[zabbix]` настраиваются свойства подключения к серверу Zabbix и могут содержаться следующие параметры:

`enabled`

Включает/отключает передачу собранных метрик на сервер Zabbix при значении, равном `True` или `False`, соответственно.

По умолчанию: `True` (вкл.)

`client`

Имя узла в Zabbix.

`address`

Адрес сервера Zabbix.

Значение по умолчанию: 127.0.0.1

`port`

Порт сервера Zabbix.

Значение по умолчанию: 10051

`timeout`

Максимальное время ожидания при подключении к серверу Zabbix (в секундах).

Значение по умолчанию: 15

[agent]

В разделе `[agent]` задаётся расположение службы mamonsu и включается возможность обращения к метрикам в командной строке:

`enabled`

Включает/отключает сбор метрик в командной строке с помощью команды `agent`.

По умолчанию: `True` (вкл.)

`host`

Адрес системы, в которой работает mamonsu.

Значение по умолчанию: 127.0.0.1

`port`

Порт, через который mamonsu принимает соединения.

Значение по умолчанию: 10052

[sender]

В разделе `[sender]` определяется размер очереди, в которой накапливаются данные, прежде чем они передаются серверу Zabbix:

`queue`

Максимальное количество собранных значений метрик, которое может накапливаться локально, прежде чем mamonsu передаст их на сервер Zabbix. После передачи накопленных данных очередь очищается.

Значение по умолчанию: 2048

Параметры вывода

[metric_log]

В разделе `[metric_log]` настраивается сохранение собранных значений метрик в локальных текстовых файлах. Этот раздел может содержать следующие параметры:

`enabled`

Включает/отключает сохранение собранных значений метрик в текстовом файле. Если этот параметр равен `True`, mamonsu создаёт файл `localhost.log` для сохранения значений метрик.

Значение по умолчанию: `False`

`directory`

Указывает каталог, в котором будут сохраняться данные метрик.

Каталог по умолчанию: `/var/log/mamonsu`

`max_size_mb`

Максимальный размер файла метрик, в мегабайтах. По достижении заданного размера этот файл переименовывается в `localhost.log.archive`, и создаётся новый файл `localhost.log`.

Значение по умолчанию: 1024

[log]

В разделе `[log]` настраивается запись сообщений в журнал mamonsu и могут содержаться следующие параметры:

`file`

Задаёт имя файла журнала; может также задавать полный путь.

`level`

Задаёт уровень отладки. Этот параметр может принимать значение `DEBUG`, `ERROR` или `INFO`.

Значение по умолчанию: `INFO`

`format`

Формат записываемых сообщений.

Значение по умолчанию: `[%(asctime)s] %(asctime)s - %(name)s - %(message)s`

Здесь `levelname` — уровень важности сообщения, `asctime` — текущее время, `name` — имя модуля, выдавшего это сообщение (либо `root`, если сообщение выдано не модулем), и `message` — собственно текст сообщения.

Параметры модулей

[plugins]

В разделе `[plugins]` описываются дополнительные модули, осуществляющие сбор метрик, и могут содержаться следующие параметры:

`enabled`

Включает/отключает дополнительные модули, осуществляющие сбор метрик, при значении равном `True` или `False`, соответственно.

Значение по умолчанию: `False`

`directory`

Задаёт каталог, содержащий дополнительные модули. При значении, равном `None`, использование дополнительных модулей отключается.

Значение по умолчанию: `/etc/mamonsu/plugins`

Если вы хотите настроить модуль, который вы дополнительно добавили в mamonsu после установки, для него нужно добавить раздел параметров в файл `agent.conf`.

Синтаксис, которому должен соответствовать этот раздел, показан в примере в Подразделе «Разделы отдельных модулей».

Разделы отдельных модулей

Все встроенные модули устанавливаются в составе mamonsu. Чтобы настроить такой модуль, найдите соответствующий ему раздел под заголовком `Individual Plugin Sections` (Разделы отдельных модулей) и измените значения параметров.

Чтобы отключить какой-либо модуль, присвойте его параметру `enabled` значение `False`, а чтобы включить — значение `True`. Эти значения задаются с учётом регистра.

В следующем примере показаны разделы, соответствующие встроенным модулям `preparedtransaction` и `pgprobackup`:

```
[preparedtransaction]
max_prepared_transaction_time = 60
interval = 60

[pgprobackup]
enabled = false
interval = 300
backup_dirs = /backup_dir1,/backup_dir2
pg_probackup_path = /usr/bin/pg_probackup-11
```

[preparedtransaction]

Этот модуль получает возраст старейшей подготовленной транзакции в секундах, а также число всех транзакций, подготовленных для двухфазной фиксации. За дополнительными сведениями обратитесь к PREPARE TRANSACTION и Разделу 49.77.

Параметр `max_prepared_transaction_time` задаёт в секундах пороговое значения для возраста подготовленной транзакции.

Параметр `interval` позволяет изменить интервал сбора метрик.

Данный модуль собирает две метрики: `pgsql.prepared.count` (число подготовленных транзакций) и `pgsql.prepared.oldest` (возраст старейшей подготовленной транзакции в секундах).

Если значение `pgsql.prepared.oldest` превышает порог, заданный параметром `max_prepared_transaction_time`, срабатывает триггер со следующим сообщением: «PostgreSQL prepared transaction is too old on {host}» (Слишком старая подготовленная транзакция PostgreSQL на {узел}).

[pgprobackup]

Этот модуль обращается к программе pg_probackup для отслеживания состояния её резервных копий, а также вычисляет объём каталогов, в которых хранятся все файлы копий.

Заметьте, что этот модуль подсчитывает общий объём всех файлов в целевом каталоге. Убедитесь в том, что в целевой каталог не попадают посторонние файлы.

Параметр `backup_dirs` задаёт разделённый запятыми список путей к каталогам, для которых будут собираться метрики.

Параметр `pg_probackup_path` задаёт путь к программе pg_probackup.

Параметр `interval` позволяет изменить интервал сбора метрик.

По умолчанию этот модуль отключён. Чтобы включить его, задайте для параметра `enabled` значение `True`.

Данный модуль собирает две метрики: `pg_probackup.dir.size[#каталог_копий]` (объём целевого каталога) и `pg_probackup.dir.error[#каталог_копий]` (ошибки копий) для каждого указанного *`каталога_копий`*.

Если какая-либо копия имеет ошибочное состояние, например `ERROR`, `CORRUPT`, `ORPHAN`, этот модуль вызывает срабатывание триггера.

Использование

Сбор и просмотр метрик

Сразу после запуска mamonsu начинает сбор метрик системы и Postgres Pro. Команда `agent` позволяет в реальном времени получать текущие значения собираемых метрик в командной строке.

Чтобы просмотреть список имеющихся метрик, воспользуйтесь командой `agent metric-list`:

```
mamonsu agent metric-list
```

Вывод этой команды включает ключ метрики, её последнее значение и время получения этого значения. Например:

```
system.memory[active]      7921004544    1564570818
system.memory[swap_cache]      868352    1564570818
pgsql.connections[idle]           6.0    1564570818
pgsql.archive_command[failed_trying_to_archive]    0    1564570818
```

Чтобы просмотреть текущее значение определённой метрики, воспользуйтесь командой `agent metric-get`:

```
mamonsu agent metric-get имя_метрики
```

Здесь *`имя_метрики`* — ключ интересующей вас метрики, представленный в выводе команды `metric-list`. Например: `pgsql.connections[idle]`.

Вы можете просмотреть графики собранных метрик в веб-интерфейсе Zabbix в разделе **Graphs** (Графики). Подробнее об использовании Zabbix рассказывается в его официальной документации по адресу *https://www.zabbix.com/documentation/current/ru/*.

Если вы решили сохранять все собираемые метрики в файле как описано в «Параметры вывода», собранные метрики можно передать на сервер Zabbix для визуализации, воспользовавшись командой upload.

Добавление дополнительных модулей

Вы можете расширить функциональность mamonsu, добавив собственные дополнительные модули следующим образом:

1. Сохраните все нужные вам дополнительные модули в одном каталоге, например, в `/etc/mamonsu/plugins`.
2. Убедитесь в том, что этот каталог указан в файле конфигурации в разделе `[plugins]`:

   ```
   [plugins]
   directory = /etc/mamonsu/plugins
   ```

3. Сгенерируйте новый шаблон Zabbix, включающий информацию дополнительных модулей:

   ```
   mamonsu export template template.xml --add-plugins=/etc/mamonsu/plugins
   ```

4. Загрузите сгенерированный `template.xml` на сервер Zabbix, как описано в Разделе «Установка и начальная настройка».

Оптимизация конфигурации Postgres Pro и системы

В зависимости от собранных значений метрик, mamonsu может настроить конфигурацию Postgres Pro и системы для оптимальной производительности.

Вы можете получить подробную информацию об оборудовании, операционной системе, использовании памяти и других характеристиках объекта мониторинга, а также конфигурацию Postgres Pro, выполнив команду:

```
mamonsu report
```

Чтобы только просмотреть предлагаемые для оптимизации изменения, но не применять их, запустите команду `tune` с ключом `--dry-run`:

```
mamonsu tune --dry-run
```

Чтобы предлагаемые изменения были применены, запустите команду `tune` без каких-либо параметров:

```
mamonsu tune
```

Вы можете исключить некоторые параметры из отчёта или отключить некоторые методы оптимизации, передав дополнительные аргументы. Полный список аргументов команд `report` и `tune` приведён в Разделе «Справка по командной строке».

Управление параметрами сервера Zabbix в командной строке

Приложение mamonsu позволяет работать с сервером Zabbix, используя командную строку. В частности, вы можете загружать файлы шаблонов на сервер Zabbix, создавать и удалять узлы и группы узлов Zabbix, привязывать шаблоны к узлам и получать последние значения метрик.

Чтобы настроить узел мониторинга в Zabbix с самого начала, вы можете выполнить следующие действия:

1. При желании укажите учётные данные в Zabbix в следующих переменных окружения в той системе, где настраивается мониторинг:

   - Задайте в переменных `ZABBIX_USER` и `ZABBIX_PASSWD` имя и пароль вашей учётной записи Zabbix, соответственно.
   - Задайте в переменной `ZABBIX_URL` адрес `http://zabbix/`

   Если этого не сделать, вам нужно будет указывать следующие параметры при каждом вызове любой подкоманды `mamonsu zabbix`:

   ```
   --url=http://zabbix/ --user=имя_пользователя_zabbix --
   password=пароль_пользователя_zabbix
   ```

2. Сгенерируйте новый файл шаблона и загрузите его на сервер Zabbix:

   ```
   mamonsu export template template.xml
   mamonsu zabbix template export template.xml
   ```

3. Создайте новую группу узлов:

   ```
   mamonsu zabbix hostgroup create имя_группы_узлов
   ```

4. Определите идентификаторы созданной группы и загруженного шаблона, назначаемые автоматически сервером Zabbix:

   ```
   mamonsu zabbix hostgroup id имя_группы_узлов
   mamonsu zabbix template id имя_шаблона
   ```

5. Создайте узел, связанный с этой группой, и привяжите к нему загруженный шаблон с помощью одной команды:

   ```
   mamonsu zabbix host create имя_узла ид_группы_узлов ид_шаблона адрес_mamonsu
   ```

Здесь *имя_узла* — имя создаваемого узла, *ид_группы_узлов* и *ид_шаблона* — уникальные идентификаторы, полученные на предыдущем шаге, а *mamonsu_address* — IP-адрес системы, в которой работает mamonsu.

Создав узел в Zabbix, произведите настройку целевой системы, как описано в Разделе «Установка и начальная настройка».

Экспорт метрик для стандартного агента Zabbix

Используя mamonsu, вы можете преобразовать определения метрик системы и Postgres Pro в формат, который поддерживает стандартный *агент Zabbix*.

В настоящее время эта функциональность имеет следующие ограничения:

- В Windows экспорт метрик `mamonsu` не поддерживается.
- Определения метрик входящего в Postgres Pro Enterprise расширения `pg_wait_sampling` и механизма CFS не преобразуются.

> **Важно**
>
> Для работы с прокси Zabbix необходимо экспортировать все параметры в формат, который поддерживается стандартным агентом Zabbix.

Чтобы использовать агент Zabbix для сбора метрик mamonsu, выполните следующее:

1. Сгенерируйте файл конфигурации, совместимый со стандартным агентом Zabbix, и поместите его в `/etc/zabbix/zabbix_agent.d/`. Вы можете добавить нужный путь перед именем файла:

   ```
   mamonsu export zabbix-parameters /etc/zabbix/zabbix_agent.d/zabbix_agent.conf
   ```

   Все имеющиеся параметры команды `export zabbix-parameters` описаны в Разделе «Справка по командной строке».
2. Сгенерируйте шаблон для стандартного агента Zabbix:

   ```
   mamonsu export zabbix-template template.xml
   ```

   Все параметры команды `export zabbix-template` описаны в Разделе «Справка по командной строке».
3. Загрузите сгенерированный шаблон на сервер Zabbix, как описано в Разделе «Установка и начальная настройка».
4. Если вы намерены собирать метрики Postgres Pro, после загрузки шаблона измените в веб-интерфейсе следующие макросы:
   - Вместо `{$PG_CONNINFO}` укажите параметры подключения к серверу Postgres Pro, за которым вы намерены наблюдать.
   - Вместо `{$PG_PATH}` укажите путь к каталогу, где установлена программа psql.

Замечания о совместимости

Если вы хотите обновить mamonsu до версии, несовместимой с предыдущей, то ваши дальнейшие действия зависят от того, нужно ли вам сохранить собранные данные метрик.

- Если вам нужно сохранить собранные данные, выполните следующее:
  1. Установите новую версию mamonsu.
  2. Сгенерируйте новый шаблон для сервера Zabbix.
  3. Если вы выполняли специальную подготовительную процедуру в предыдущей версии mamonsu, запустите команду `bootstrap` ещё раз.

4. Загрузите новый шаблон на сервер Zabbix.
5. Переименуйте узел, для которого вы хотите сохранить собранные данные, и оставьте его привязку к старому шаблону.
6. Создайте новый узел для той же системы и привяжите к нему новый шаблон.
7. Перезапустите службу mamonsu. Она будет собирать данные для нового узла. Старый узел больше не будет использоваться, но собранные для него данные будут по-прежнему доступны.

- Если вам не нужно сохранять собранные данные, выполните следующее:

1. Установите новую версию mamonsu.
2. Сгенерируйте новый шаблон для сервера Zabbix.
3. Если вы выполняли специальную подготовительную процедуру в предыдущей версии mamonsu, запустите команду `bootstrap` ещё раз.
4. Загрузите новый шаблон на сервер Zabbix.
5. В параметрах узла Zabbix привяжите к узлу новый шаблон вместо старого.
6. Перезапустите службу mamonsu, чтобы начался сбор данных. Все данные, собранные ранее, будут потеряны.

pgbouncer

pgbouncer — пул соединений Postgres Pro

Синтаксис

В системах Linux:

`pgbouncer` [-d] [-R] [-v] [-u *`пользователь`*] pgbouncer.ini

`pgbouncer` -V | -h

В Windows:

`pgbouncer` [-v] [-u *`пользователь`*] pgbouncer.ini

`pgbouncer` -V | -h

Для использования pgbouncer в виде службы Windows есть дополнительные аргументы:

`pgbouncer.exe` --regservice pgbouncer.ini

`pgbouncer.exe` --unregservice pgbouncer.ini

Описание

pgbouncer — это программа, управляющая пулом соединений Postgres Pro. Любое конечное приложение может подключиться к pgbouncer, как если бы это был непосредственно сервер Postgres Pro, и pgbouncer создаст подключение к реальному серверу, либо задействует одно из ранее установленных подключений.

Предназначение pgbouncer — минимизировать издержки, связанные с установлением новых подключений к Postgres Pro.

Чтобы не нарушать семантику транзакций при переключении подключений, pgbouncer поддерживает несколько видов пулов:

Пул сеансов

Наиболее корректный метод. Когда клиент подключается, ему назначается одно серверное подключение на всё время, пока клиент остаётся подключённым. Когда клиент отключается, это подключение к серверу возвращается в пул. Этот метод работает по умолчанию.

Пул транзакций

Подключение к серверу назначается клиенту только на время транзакции. Когда pgbouncer замечает, что транзакция завершена, это подключение возвращается в пул.

Пул операторов

Наиболее агрессивный метод. Подключение к серверу будет возвращаться в пул сразу после завершения каждого запроса. Транзакции с несколькими операторами в этом режиме запрещаются, так как они не будут работать.

Административный интерфейс pgbouncer состоит из нескольких новых команд `SHOW`, доступных при подключении к специальной «виртуальной» базе данных `pgbouncer`.

Быстрый запуск

Базовая настройка и использование демонстрируются ниже.

1. Создайте файл `pgbouncer.ini`. Подробнее он описывается на странице man `pgbouncer(5)`. Например:

```
[databases]
template1 = host=localhost dbname=template1 auth_user=someuser

[pgbouncer]
listen_port = 6432
listen_addr = localhost
auth_type = md5
auth_file = userlist.txt
logfile = pgbouncer.log
pidfile = pgbouncer.pid
admin_users = someuser
```

2. Создайте файл `userlist.txt` со списком пользователей, которым разрешено подключение:

```
"некоторый_пользователь" "его_пароль_на_сервере"
```

3. Запустите pgbouncer:

```
$ pgbouncer -d pgbouncer.ini
```

> **Примечание**
>
> Эта команда не работает в системах Windows. Вместо этого pgbouncer должен запускаться в виде службы, которую необходимо сначала зарегистрировать следующим образом:
>
> `pgbouncer --regservice`

4. Сделайте так, чтобы ваше приложение (или клиент `psql`) подключалось к pgbouncer, а не к серверу Postgres Pro непосредственно:

```
$ psql -p 6432 -U someuser template1
```

5. Для управления pgbouncer подключитесь к специальной административной базе данных pgbouncer и выполните `SHOW HELP;`, чтобы ознакомиться с его справкой:

```
$ psql -p 6432 -U someuser pgbouncer
pgbouncer=# SHOW HELP;
NOTICE:  Console usage
DETAIL:
  SHOW [HELP|CONFIG|DATABASES|FDS|POOLS|CLIENTS|SERVERS|SOCKETS|LISTS|VERSION|...]
  SET key = arg
  RELOAD
  PAUSE
  SUSPEND
  RESUME
  SHUTDOWN
  [...]
```

6. Если вы вносили изменения в файл `pgbouncer.ini`, его можно перезагрузить командой:

```
pgbouncer=# RELOAD;
```

Параметры

`-d, --daemon`

Запустить в фоновом режиме. Без этого указания процесс будет работать на переднем плане. При запуске в фоновом режиме необходимо явное указание параметра `pidfile`, а также `logfile` или `syslog`. В фоновом режиме сообщения не будут записываться в stderr.

> **Примечание**
>
> Это не работает в Windows, там pgbouncer нужно запускать в виде службы.

`-R, --reboot`

Выполнить перезапуск на «лету». При этом pgbouncer подключается к работающему процессу, забирает у него открытые сокеты и начинает использовать их. Если активного процесса нет, он запускается в обычном режиме.

> **Примечание**
>
> Это работает, только если ОС поддерживает сокеты Unix и в конфигурации определён параметр `unix_socket_dir`. Не работает в Windows. Также при этом не поддерживаются подключения TLS, они сбрасываются.

`-u` *`пользователь`*`, --user` *`пользователь`*

Переключиться на заданного пользователя при запуске.

`-v, --verbose`

Увеличить детализацию вывода. Может использоваться неоднократно.

`-q, --quiet`

Приглушить вывод: не выводить ничего в stderr. Это не влияет на уровень выводимых сообщений, а только отключает вывод в stderr. Предназначено для применения в скриптах `init.d`.

`-V, --version`

Вывести версию.

`-h, --help`

Вывести короткую справку.

`--regservice`

Win32: Зарегистрировать pgbouncer в качестве службы Windows. Имя службы будет определяться значением параметра конфигурации `service_name`.

`--unregservice`

Win32: Разрегистрировать службу Windows.

Административная консоль

Эта консоль доступна при обычном подключении к базе pgbouncer:

```
$ psql -p 6432 pgbouncer
```

Подключаться к этой консоли разрешено только пользователям, перечисленным в параметрах конфигурации `admin_users` или `stats_users`. (За исключением режима `auth_mode=any`, когда в качестве `stats_user` может подключиться любой пользователь.)

Кроме того, пользователю `pgbouncer` разрешено подключение без пароля, если это подключение устанавливается через сокет Unix и у клиента тот же uid пользователя Unix, что и у работающего процесса.

Команды вывода информации

Команды `SHOW` выводят полезную информацию. Каждая команда описана ниже.

SHOW STATS

Выводит статистику. Выводимые этой и связанными командами суммарные значения, накапливаемые с момента запуска pgbouncer, и средние значения обновляются с периодичностью `stats_period`.

`database`

База данных, для которой представлена статистика.

`total_xact_count`

Общее число транзакций SQL, прошедших через pgbouncer.

`total_query_count`

Общее число SQL-запросов, прошедших через pgbouncer.

`total_received`

Общий объём сетевого трафика (в байтах), который получил pgbouncer.

`total_sent`

Общий объём сетевого трафика (в байтах), который передал pgbouncer.

`total_xact_time`

Общее время (в микросекундах), в течение которого pgbouncer использовал подключения к Postgres Pro, выполняя запросы или простаивая.

`total_query_time`

Общее время (в микросекундах), в течение которого pgbouncer активно использовал подключения к Postgres Pro, выполняя запросы.

`total_wait_time`

Время, в течение которого клиенты ожидали ответов сервера (в микросекундах).

`avg_xact_count`

Среднее число транзакций в секунду за последний период статистики.

`avg_query_count`

Среднее число запросов в секунду за последний период статистики.

`avg_recv`

Средняя скорость получения данных от клиентов (байт в секунду).

`avg_sent`

Средняя скорость передачи данных клиентам (байт в секунду).

`avg_xact_time`

Средняя длительность транзакции (в микросекундах).

`avg_query_time`

Средняя длительность запроса (в микросекундах).

`avg_wait_time`

Среднее время ожидания ответов сервера в течение секунды (в микросекундах).

SHOW STATS_TOTALS

Выводит подмножество результатов `SHOW STATS`, включающее только суммарные значения (`total_`).

SHOW STATS_AVERAGES

Выводит подмножество результатов `SHOW STATS`, включающее только средние значения (`avg_`).

SHOW TOTALS

Выводит ту же статистику, что `SHOW STATS`, но по всем базам данных в целом.

SHOW SERVERS

`type`

«S» для серверов

`user`

Имя пользователя, с которым pgbouncer подключается к серверу.

`database`

Имя базы данных.

`state`

Состояние подключения pgbouncer к серверу, один из вариантов: `active`, `idle`, `used`, `tested` или `new`.

`addr`

IP-адрес сервера Postgres Pro.

`port`

Порт сервера Postgres Pro.

`local_addr`

Исходный адрес подключения на локальной машине.

`local_port`

Исходный порт подключения на локальной машине.

`connect_time`

Время установления подключения.

`request_time`

Время выдачи последнего запроса.

`wait`

Текущая длительность ожидания (в секундах).

`wait_us`

Дробная часть текущей длительности ожидания (в микросекундах).

`close_needed`

1, если соединение будет закрыто при ближайшей возможности в связи с выполнением команды `RECONNECT` либо в связи с изменением параметров соединения, вызванным перезагрузкой файла конфигурации или изменениями в DNS.

`ptr`

Адрес внутреннего объекта для данного подключения. Используется как уникальный идентификатор.

`link`

Адрес клиентского подключения, с которым связан сервер.

`remote_pid`

Идентификатор обслуживающего серверного процесса (PID). Когда подключение выполняется через сокет Unix и ОС может выдать PID процесса, это PID, полученный от ОС. В противном случае этот идентификатор извлекается из пакета для отмены, переданного сервером; это будет интересующий PID при подключении к серверу Postgres Pro, но если подключение обслуживает другой pgbouncer, это будет случайное число.

`tls`

Информация о TLS-подключении; пустая строка, если TLS не используется.

SHOW CLIENTS

`type`

«C» для клиентов.

`user`

Пользователь, подключённый со стороны клиента.

`database`

Имя базы данных.

`state`

Состояние клиентского подключения: `active` или `waiting`.

`addr`

IP-адрес клиента.

`port`

Порт клиента.

`local_addr`

Конечный адрес подключения на локальной машине.

`local_port`

Конечный порт подключения на локальной машине.

`connect_time`

Время установления подключения.

`request_time`

Время последнего запроса клиента.

`wait`

Текущая длительность ожидания (в секундах).

`wait_us`

Дробная часть текущей длительности ожидания (в микросекундах).

`close_needed`

Не используется для клиентов.

`ptr`

Адрес внутреннего объекта для данного подключения. Используется как уникальный идентификатор.

`link`

Адрес серверного подключения, с которым связан клиент.

`remote_pid`

Идентификатор процесса (PID), в случае, если клиент подключается через сокет UNIX и ОС может выдать этот идентификатор.

`tls`

Информация о TLS-подключении; пустая строка, если TLS не используется.

SHOW POOLS

Новый пул создаётся для каждой пары сущностей (база данных, пользователь).

`database`

Имя базы данных.

`user`

Имя пользователя.

`cl_active`

Число клиентских подключений, которые связаны с подключениями к серверу и могут обрабатывать запросы.

`cl_waiting`

Число клиентских подключений, которые отправили запросы, но ещё не получили подключения к серверу.

`cl_cancel_req`

Число клиентских подключений, для которых серверу в данный момент передаются команды отмены запроса.

`sv_active`

Число серверных подключений, связанных с клиентами.

`sv_idle`

Число серверных подключений, которые не используются и могут немедленно задействоваться для запросов клиентов.

`sv_used`

Число серверных подключений, которые простаивают дольше `server_check_delay`, поэтому перед повторным использованием для них нужно выполнять `server_check_query`.

`sv_tested`

Число серверных подключений, для которых в данный момент выполняются запросы `server_reset_query` или `server_check_query`.

`sv_login`

Число серверных подключений, через которые в данный момент выполняется вход на сервер.

`maxwait`

Показывает, как долго ожидает в очереди самый первый клиент (в секундах). Если это число начинает увеличиваться, значит текущий пул серверов не справляется с запросами достаточно быстро. Причиной тому может быть перегруженный сервер, либо просто слишком маленький размер пула (параметр `pool_size`).

`maxwait_us`

Дробная часть максимального времени ожидания (в микросекундах).

`pool_mode`

Действующий режим пула.

SHOW LISTS

Показывает следующие внутренние сведения, в столбцах (не строках):

`databases`

Число баз данных.

`users`

Число пользователей.

`pools`

Число пулов.

`free_clients`

Число свободных клиентов.

`used_clients`

Число активных клиентов.

`login_clients`

Число клиентов в состоянии входа (`login`).

`free_servers`

Число свободных серверов.

`used_servers`

Число задействованных серверов.

`dns_names`

Количество имён DNS в кеше.

`dns_zones`

Количество зон DNS в кеше.

`dns_queries`

Количество выполняющихся запросов к DNS.

`dns_pending`

Не используется.

SHOW USERS

`name`

Имя пользователя.

`pool_mode`

Переопределение `pool_mode` для пользователя либо `NULL`, если должен использоваться режим по умолчанию.

SHOW DATABASES

`name`

Имя настроенной записи базы данных.

`host`

Компьютер, к которому подключается pgbouncer.

`port`

Порт, к которому подключается pgbouncer.

`database`

Реальное имя базы данных, к которой подключается pgbouncer.

`force_user`

Когда пользователь указан в строке соединения, подключение между pgbouncer и Postgres Pro должно устанавливаться от его имени, вне зависимости от пользователя на стороне клиента.

`pool_size`

Максимальное число серверных подключений.

`min_pool_size`

Минимальное число серверных подключений.

`reserve_pool`

Максимальное число дополнительных подключений для этой базы данных.

`pool_mode`

Переопределение pool_mode для базы данных либо `NULL`, если должен использоваться режим по умолчанию.

`max_connections`

Максимально возможное число подключений для этой базы, установленное либо глобально, параметром `max_db_connections`, либо на уровне базы.

`current_connections`

Текущее число подключений для этой базы.

`paused`

1, если база данных находится в состоянии паузы, иначе — 0.

`disabled`

1, если база данных находится в отключённом состоянии, иначе — 0.

SHOW FDS

Внутренняя команда, которая показывает список файловых дескрипторов и их внутреннее состояние.

При подключении пользователя с именем `pgbouncer` через сокет Unix из процесса с UID, совпадающим с UID текущего процесса, ему передаются реальные файловые дескрипторы. Это применяется для выполнения перезагрузки «на лету».

> **Примечание**
>
> В Windows это не работает.

Эта команда также блокирует внутренний цикл событий, так что её не следует выполнять, когда pgbouncer используется.

`fd`

Числовое значение файлового дескриптора (ФД).

`task`

Предназначение; возможны следующие варианты: `pooler`, `client` или `server`.

`user`

Пользователь подключения, занимающего этот ФД.

`database`

База данных подключения, занимающего этот ФД.

`addr`

IP-адрес подключения, занимающего данный ФД; `unix`, если это сокет Unix.

`port`

Порт подключения, занимающего данный ФД.

`cancel`

Ключ отмены для данного подключения.

`link`

Файловый дескриптор ответной стороны сервера/клиента. `NULL`, если подключение простаивает.

SHOW SOCKETS, SHOW ACTIVE_SOCKETS

Выводит низкоуровневую информацию обо всех или только активных сокетах. В вывод включается информация, выводимая командами `SHOW CLIENTS` и `SHOW SERVERS`, а также дополнительные сведения низкого уровня.

SHOW CONFIG

Выводит текущие параметры конфигурации, по одному в строке со следующими столбцами:

`key`

Имя переменной конфигурации.

`value`

Значение переменной конфигурации.

`default`

Значение по умолчанию.

`changeable`

Значение `yes` или `no`, показывающее, можно ли изменить переменную во время работы. Если значение `no`, переменная может быть изменена только при перезагрузке. Чтобы изменить переменную во время работы, воспользуйтесь командой `SET`.

SHOW MEM

Выводит низкоуровневую информацию о размере различных блоков памяти, выделенных для внутреннего использования. Эта информация имеет динамическую природу и может меняться.

SHOW DNS_HOSTS

Выводит имена узлов, содержащиеся в кеше DNS.

`hostname`

Имя узла.

`ttl`

Сколько секунд остаётся до очередного внешнего поиска.

`addrs`

Список адресов, разделённых запятыми.

SHOW DNS_ZONES

Показывает зоны DNS в кеше.

`zonename`

Имя зоны.

`serial`

Текущий серийный номер.

`count`

Имена узлов, относящихся к этой зоне.

SHOW VERSION

Показывает строку с версией pgbouncer.

Команды управления процессом

PAUSE [*бд*]

pgbouncer пытается отключиться ото всех серверов, сначала ожидая завершения всех запросов. Выполнение этой команды заканчивается, только когда завершаются все запросы. Эта команда должна применяться во время перезапуска базы данных.

Если указано имя базы данных, будет приостановлена работа только с ней.

Новые подключения клиентов к базе, поставленной на паузу, будут находиться в состоянии ожидания, пока не будет выполнена команда `RESUME`.

DISABLE *бд*

Запрещает любые новые подключения клиентов к указанной базе данных.

ENABLE *бд*

Разрешает новые подключения клиентов после предыдущей команды `DISABLE`.

RECONNECT *бд*

Закрывает все открытые подключения к указанной базе (или ко всем базам) по мере их освобождения в соответствии с режимом пула, не дожидаясь окончания времени жизни подключений. При этом немедленно могут быть установлены новые подключения к серверу, согласно текущим параметрам пула.

Эта команда может быть полезна, когда изменяется конфигурация подключения к серверу, например, нужно постепенно переключиться на новый сервер. Эту команду не нужно выполнять, если вы меняете строку соединения в `pgbouncer.ini` и перезагружаете конфигурацию (см. `RELOAD`) или когда меняются адреса в DNS, так как равнозначное действие будет выполнено автоматически. Эта команда может быть полезна, только если подключения pgbouncer маршрутизируются какими-то внешними средствами.

После выполнения этой команды возможен длительный период, когда будут существовать и старые подключения, и новые. Это может иметь значение только при переключении трафика между читающими репликами или при переключении между узлами в конфигурации с несколькими ведущими. Если необходимо переключить все подключения сразу, рекомендуется использовать команду `PAUSE`. Чтобы закрыть подключения экстренно (например, когда произвести переключение нужно не постепенно, а срочно), вы можете воспользоваться командой `KILL`.

KILL *бд*

Немедленно закрывает все клиентские и серверные подключения к указанной базе данных, деактивируя её.

Новые подключения клиентов к деактивированной базе будут находиться в состоянии ожидания, пока не будет выполнена команда `RESUME`.

SUSPEND

Все буферы сокетов очищаются и pgbouncer прекращает принимать данные через них. Выполнение этой команды заканчивается, только когда все буферы будут очищены. Эта команда должна применяться, когда pgbouncer перезагружается «на лету».

Новые подключения клиентов к приостановленной базе будут находиться в состоянии ожидания, пока не будет выполнена команда `RESUME`.

RESUME [*бд***]**

Восстанавливает работу после предыдущей команды `KILL`, `PAUSE` или `SUSPEND`.

SHUTDOWN

Приводит к завершению процесса pgbouncer.

RELOAD

Указывает процессу pgbouncer перезагрузить файл конфигурации и обновить значения изменяемых параметров.

PgBouncer отслеживает изменения в файле конфигурации, затрагивающие подключения к базе данных. Существующие подключения со старыми параметрами будут закрыты при ближайшем освобождении этих подключений (в соответствии с режимом пула), а новые подключения к серверу немедленно начнут использовать изменившиеся параметры соединения.

WAIT_CLOSE [*бд***]**

Ожидает, пока для всех серверных подключений к указанной базе (или ко всем базам) не сбросится признак `close_needed` (см. Подраздел «SHOW SERVERS»). Эту команду можно вызвать после `RECONNECT` или `RELOAD`, чтобы, например, в скриптах переключения узлов, дождаться применения изменений конфигураций в полном объёме.

Другие команды

SET *ключ = аргумент*

Изменяет параметр конфигурации (см. также Подраздел «SHOW CONFIG»). Например:

```
SET log_connections = 1;
SET server_check_query = 'select 2';
```

(Заметьте, что эта команда выполняется в консоли администратора pgbouncer и задаёт параметры pgbouncer. Команда `SET`, выполненная в другой базе данных, будет передана на выполнению серверу Postgres Pro, как и любая другая SQL-команда.)

Сигналы

`SIGHUP`

Перезагрузка конфигурации. Равносильно выполнению команды `RELOAD` в консоли.

`SIGINT`

Безопасное отключение. Равносильно выполнению команд `PAUSE` и `SHUTDOWN` в консоли.

`SIGTERM`

Немедленное отключение. Равносильно выполнению команды `SHUTDOWN` в консоли.

`SIGUSR1`

Равносильно выполнению команды `PAUSE` в консоли.

`SIGUSR2`

Равносильно выполнению команды `PAUSE` в консоли.

Параметры libevent

Из документации libevent:

> Поддержку `epoll`, `kqueue`, `devpoll`, `poll` или `select` можно отключить, установив переменную окружения `EVENT_NOEPOLL`, `EVENT_NOKQUEUE`, `EVENT_NODEVPOLL`, `EVENT_NOPOLL` или `EVENT_NOSELECT`, соответственно.
>
> Если установить переменную окружения `EVENT_SHOW_METHOD`, `libevent` покажет текущий выбранный метод уведомлений в ядре.

Файл конфигурации `pgbouncer.ini`

Файл конфигурации имеет формат `ini`-файла. Названия разделов записываются между `[` и `]`. Строки, начинающиеся с `;` или `#`, считаются комментариями и игнорируются. Символы `;` и `#`, встречающиеся не в начале строки, не распознаются.

Общие параметры

`logfile`

Указывает имя файла журнала. Для работы в режиме демона (`-d`) должен быть задан либо этот параметр, либо `syslog`. Файл журнала остаётся открытым, так что после смены имени для прокрутки следует выполнить `kill -HUP` или `RELOAD;` в консоли. В Windows необходимо остановить и вновь запустить службу.

Обратите внимание, что параметр `logfile` сам по себе не отключает вывод сообщений в stderr. Отключить его можно, воспользовавшись параметром командной строки `-q` или `-d`.

По умолчанию: не задано.

`pidfile`

Указывает имя файла PID. Без этого указания работа в режиме демона (`-d`) не допускается.

По умолчанию: не задано.

`listen_addr`

Указывает список адресов, по которым должны приниматься TCP-подключения. Вы можете указать `*`, что будет означать «принимать по всем адресам». Когда этот параметр не задан, принимаются только подключения через Unix-сокеты.

Адреса могут задаваться числами (в формате IPv4/IPv6) или именами.

По умолчанию: не задано.

`listen_port`

Номер принимающего порта. Задаётся и для сокетов TCP, и для Unix-сокетов.

По умолчанию: 6432

`unix_socket_dir`

Указывает расположение сокетов Unix. Действует и для принимающего сокета, и для подключений к серверу. Если задана пустая строка, сокеты Unix отключаются. При указании значения, начинающегося с `@`, будет создан сокет Unix в абстрактном пространстве имён (в настоящее время поддерживается в Linux и Windows).

Чтобы работала перезагрузка «на лету» (`-R`), необходимо настроить сокет Unix, который должен находиться в пространстве имён файловой системы.

По умолчанию: /tmp (в Windows — пустая строка)

`unix_socket_mode`

Режим файловой системы для сокета Unix. Игнорируется для сокетов в абстрактном пространстве имён. Не поддерживается в Windows.

По умолчанию: 0777

`unix_socket_group`

Имя группы для сокета Unix. Игнорируется для сокетов в абстрактном пространстве имён. Не поддерживается в Windows.

По умолчанию: не задано.

`user`

Если задан, определяет, на какого пользователя Unix нужно переключиться после запуска. Работает, только если pgbouncer запускается от имени root или уже запущен от имени заданного пользователя.

Не поддерживается в Windows.

По умолчанию: не задано.

`auth_file`

Имя файла, из которого будут загружаться имена и пароли пользователей. За подробностями обратитесь к Подразделу «Формат файла аутентификации».

По умолчанию: не задано.

auth_hba_file

Файл конфигурации HBA, который используется, когда параметр auth_type равен hba.

По умолчанию: не задано.

auth_type

Определяет, как аутентифицировать пользователей.

pam

Для проверки подлинности пользователей используется инфраструктура PAM (Pluggable Authentication Modules, Подключаемые модули аутентификации). Файл auth_file игнорируется. Этот метод несовместим с базами данных, для которых используется auth_user. Инфраструктуре PAM в качестве имени службы передаётся pgbouncer. PAM в файле конфигурации HBA не поддерживается.

hba

Фактический тип аутентификации загружается из auth_hba_file. Это позволяет применять разные методы аутентификации для разных вариантов доступа. Например: для подключений через сокет Unix применять метод peer, а для TCP — TLS.

cert

Клиент должен подключаться по соединению TLS с действительным клиентским сертификатом. Имя пользователя берётся из поля CommonName (Общее имя) сертификата.

md5

Применять проверку пароля по хешу MD5. Этот метод аутентификации выбирается по умолчанию. Файл auth_file может содержать как зашифрованные MD5, так и открытые пароли. Даже при выборе md5, если пароль пользователя задан для метода SCRAM, автоматически будет применяться проверка по алгоритму SCRAM.

scram-sha-256

Применять проверку пароля по алгоритму SCRAM-SHA-256. Заданный параметром auth_file файл должен содержать зашифрованные SCRAM или открытые пароли. Учтите, что зашифрованные SCRAM пароли могут использоваться только для проверки пароли клиентов, но не для входа на сервер. Чтобы использовать SCRAM для серверных подключений, пароли необходимо задать открытым текстом.

plain

По каналу передаётся пароль в открытом тексте. Устаревший вариант.

trust

Аутентификация не выполняется. Тем не менее имя пользователя должно присутствовать в auth_file.

any

Подобен методу trust, но переданное имя пользователя игнорируется. Требует, чтобы для всех баз данных было настроено подключение заданного пользователя. Кроме того, база данных консоли допускает подключение любого пользователя в качестве администратора.

auth_query

Запрос для извлечения пароля пользователя из базы данных.

Для прямого доступа к pg_shadow требуются права администратора. Поэтому рекомендуется, чтобы обычный пользователь обращался к ней, вызывая функцию SECURITY DEFINER (с контекстом безопасности определившего).

Заметьте, что этот запрос выполняется в целевой базе данных, так что если в нём используются функции, они должны быть установлены в каждой базе.

По умолчанию: `SELECT usename, passwd FROM pg_shadow WHERE usename=$1`

`auth_user`

Если задан параметр `auth_user`, пользователи, не описанные в файле `auth_file`, будут проверяться запросом `auth_query` по таблице `pg_shadow` в базе данных `auth_user`. Пароль пользователя `auth_user` будет взят из файла `auth_file`. (Если для пользователя `auth_user` не требуется пароль, задавать его в `auth_file` не нужно.)

Для прямого доступа к `pg_shadow` требуются права администратора. Поэтому рекомендуется, чтобы обычный пользователь обращался к ней, вызывая функцию SECURITY DEFINER (с контекстом безопасности определившего).

По умолчанию: не задано.

`pool_mode`

Указывает, когда подключение к серверу могут повторно использовать другие клиенты.

`session`

Сервер возвращается в пул после отключения клиента. Это вариант по умолчанию.

`transaction`

Сервер возвращается в пул после завершения транзакции.

`statement`

Сервер возвращается в пул после завершения запроса. В этом режиме не допускаются транзакции, охватывающие несколько операторов.

`max_client_conn`

Максимально допустимое число клиентских подключений. При увеличении должно также увеличиваться ограничение на число файловых дескрипторов. Заметьте, что фактическое число занятых файловых дескрипторов будет больше чем `max_client_conn`. Если каждый пользователь подключается к серверу под своим именем, теоретически возможный максимум равен:

```
max_client_conn + (max pool_size * total databases * total users)
```

Если пользователь задан в строке соединения (все пользователи подключаются под одним именем), теоретический максимум равен:

```
max_client_conn + (max pool_size * total databases)
```

Теоретический максимум не должен достигаться никогда, если только кто-то намеренно не предпримет специальные меры для этого. Тем не менее это значит, что число файловых дескрипторов должно ограничиваться довольно большим числом.

Поищите `ulimit` в руководстве man в вашей системе. Замечание: ограничение `ulimit` неприменимо в среде Windows.

По умолчанию: 100

`default_pool_size`

Сколько подключений к серверу возможно для пары пользователь/база. Может быть переопределено в конфигурации базы данных.

По умолчанию: 20

`min_pool_size`

Добавить в пул дополнительные соединения, если число активных подключений меньше этого числа. Даёт положительный эффект, когда нагрузка появляется внезапно после периода простоя. Это значение будет ограничено размером пула.

По умолчанию: 0 (отключено)

`reserve_pool_size`

Число дополнительно разрешённых подключений в пуле (см. `reserve_pool_timeout`). При 0 резерв отсутствует.

По умолчанию: 0 (отключено)

`reserve_pool_timeout`

Если клиент не обслуживается заданное число секунд, pgbouncer задействует дополнительные подключения из резервного пула. При 0 это не происходит.

По умолчанию: 5.0

`max_db_connections`

Не допускать больше заданного числа серверных подключений к базе данных (независимо от пользователя). При этом учитывается база данных pgbouncer, к которой подключён клиент, а не база данных Postgres Pro исходящего подключения. Данный предел можно установить для каждой базы данных в разделе `[databases]`.

Обратите внимание, что когда предел достигается, закрытие подключения клиента в одном пуле не позволяет немедленно установить другое подключение к серверу через другой пул, так как подключение первого пула по-прежнему открыто. Когда сервер закроет его (по тайм-ауту неактивности), новое подключение будет немедленно установлено для ожидающего пула.

По умолчанию: 0 (нет ограничения)

`max_user_connections`

Не допускать больше заданного числа подключений к серверу для пользователя (независимо от базы данных). При этом учитывается пользователь pgbouncer, связанный с пулом, — это может быть либо пользователь, указанный для подключения к серверу, либо, в отсутствие этого указания, пользователь, от имени которого подключился клиент. Этот предел можно установить для каждого пользователя в разделе `[users]`.

Обратите внимание, что когда предел достигается, закрытие подключения клиента в одном пуле не позволяет немедленно установить другое подключение к серверу через другой пул, так как подключение первого пула по-прежнему открыто. Когда сервер закроет его (по тайм-ауту неактивности), новое подключение будет немедленно установлено для ожидающего пула.

По умолчанию: 0 (нет ограничения)

`server_round_robin`

По умолчанию pgbouncer повторно использует подключения сервера в порядке LIFO (последнее пришло, первое ушло), так что основная загрузка распределяется по нескольким последним соединениям. Это даёт наибольшую производительность, если базу данных обслуживает один сервер. Но если за IP-адресом базы данных скрывается балансировщик TCP-соединений, лучше, если pgbouncer будет использовать подключения в данном режиме, обеспечивая таким образом равномерную нагрузку.

По умолчанию: 0

`ignore_startup_parameters`

По умолчанию pgbouncer принимает только параметры, которые он может отслеживать в стартовых пакетах: `client_encoding`, `datestyle`, `timezone` и `standard_conforming_strings`.

Все другие параметры вызывают ошибку. Чтобы принимались и другие параметры, их нужно указать здесь, чтобы pgbouncer знал, что они обрабатываются администратором и их можно игнорировать.

По умолчанию: пустая строка

`disable_pqexec`

Отключает протокол простых запросов (PQexec). В отличие от протокола расширенных запросов, этот протокол допускает указание нескольких запросов в одном пакете, что оставляет место для атак с SQL-инъекцией. Отключение этого протокола может улучшить безопасность. Разумеется, это означает, что при этом смогут работать только клиенты, которые используют исключительно протокол расширенных запросов.

По умолчанию: 0

`application_name_add_host`

Добавляет адрес компьютера и порт клиента к имени приложения, задаваемого при установлении подключения. Это помогает идентифицировать источник плохих запросов и т. п. Это выполняется, только когда подключение устанавливается. Если свойство `application_name` будет изменено позднее командой `SET`, pgbouncer его уже не поменяет.

По умолчанию: 0

`conffile`

Показывает расположение текущего файла конфигурации. При изменении этого параметра pgbouncer будет использовать другой файл конфигурации после команд `RELOAD` / `SIGHUP`.

По умолчанию: файл, заданный в командной строке

`service_name`

Используется при регистрации службы win32.

По умолчанию: pgbouncer

`job_name`

Псевдоним `service_name`.

`stats_period`

Определяет, с какой периодичностью (в секундах) будут пересчитываться средние значения, выводимые различными командами `SHOW`, и как часто агрегированная статистика будет записываться в журнал (но см. `log_stats`).

По умолчанию: 60

Параметры журнала

`syslog`

Включает/отключает запись в syslog. В Windows вместо этого используется журнал событий.

По умолчанию: 0

`syslog_ident`

Имя, с которым события передаются в syslog.

По умолчанию: `pgbouncer` (имя программы)

`syslog_facility`

Субъект, который будет указываться в событиях, отправляемых в syslog. Возможные варианты: `auth`, `authpriv`, `daemon`, `user`, `local0-7`.

По умолчанию: daemon

`log_connections`

Фиксировать в журнале успешные подключения.

По умолчанию: 1

`log_disconnections`

Фиксировать отключения с указаниями их причин.

По умолчанию: 1

`log_pooler_errors`

Фиксировать сообщения об ошибках, которые pgbouncer передаёт клиентам.

По умолчанию: 1

`log_stats`

Записывать агрегированную статистику в журнал, с периодичностью `stats_period`. Это может быть лишним, если ту же статистику запрашивают внешние средства мониторинга, вызывая команды `SHOW`.

По умолчанию: 1

`verbose`

Увеличивает уровень детализации. Соответствует ключу `-v` в командной строке. Указание `-v -v` в командной строке равносильно записи `verbose=2` в конфигурации.

По умолчанию: 0

Управление доступом к консоли

`admin_users`

Разделённый запятыми список пользователей базы данных, которым разрешено подключаться к консоли и выполнять в ней любые команды. Игнорируется с `auth_type`, равным `any`, так как в этом случае любой пользователь может подключаться как администратор.

По умолчанию: пустая строка

`stats_users`

Разделённый запятыми список пользователей баз данных, которым разрешено подключаться к консоли и выполнять команды только на чтение, а именно все команды `SHOW`, за исключением `SHOW FDS`.

По умолчанию: пустая строка

Проверки активности соединений, тайм-ауты

`server_reset_query`

Запрос, посылаемый серверу при освобождении подключения, прежде чем оно станет доступно другим клиентам. В этот момент никакая транзакция не выполняется, так что запрос не должен включать команды `ABORT` или `ROLLBACK`.

Предполагается, что этот запрос очистит все изменения в состоянии сеанса базы данных, чтобы следующий клиент получил подключение в определённом состоянии. По умолчанию выполняется команда `DISCARD ALL`, которая очищает всё, но при этом следующему клиенту не остаётся никакого кешированного состояния. Её можно поменять на более мягкую, например `DEALLOCATE ALL`, просто освобождающую подготовленные операторы (если работа приложения не нарушается, когда какое-то состояние сохраняется).

Когда применяется пул транзакций, `server_reset_query` не действует, так как клиенты не должны использовать никакие свойства сеансов, потому что каждая транзакция завершается в отдельном соединении и таким образом получает разные состояния сеанса.

По умолчанию: DISCARD ALL

`server_reset_query_always`

Определяет, должен ли запрос `server_reset_query` выполняться во всех режимах пула. Когда этот параметр отключён (по умолчанию), `server_reset_query` будет запускаться только в режиме пула сеансов. Соединениям в режиме пула транзакций не должен требоваться запрос сброса состояния.

Этот параметр предназначен для аномальных конфигураций, в которых приложения не вполне корректно используют свойства сеансов, подключаясь к пулу транзакций pgbouncer. Он позволяет сменить недетерминированное неправильное поведение на детерминированное (но всё же неправильное) — клиенты всегда теряют своё состояние после каждой транзакции.

По умолчанию: 0

`server_check_delay`

Определяет, как долго должны сохраняться освобождаемые подключения в состоянии готовности к повторному использованию, без запуска запроса проверки подключения. При значении 0 этот запрос запускается всегда.

По умолчанию: 30.0

`server_check_query`

Простой холостой запрос, проверяющий, сохраняется ли подключение к серверу.

Если это пустая строка, проверка соединения отключается.

По умолчанию: SELECT 1;

`server_fast_close`

Определяет, должен ли сервер в режиме пула сеансов отключаться немедленно либо после завершения текущей транзакции, если он находится в режиме `close_needed` (который включается командами `RECONNECT` и `RELOAD` или при изменениях в DNS), или необходимо дожидаться завершения сеанса. Режимы пула запросов и транзакций так и работают, поэтому включение этого параметра на эти режимы не влияет.

Если этот параметр включён, то есть подключение к серверу закрывается до завершения сеанса клиента, клиентское подключение также закрывается. Тем самым гарантируется, что прерывание сеанса не останется незамеченным для клиента.

Этот параметр позволяет ускорить вступление в силу изменений параметров соединений в случае использования пула сеансов и существования долгоживущих сеансов. Недостатком его использования может быть то, что клиентские сеансы могут прерываться при изменении конфигурации, так что клиентские приложения должны уметь переподключаться и восстанавливать состояние сеанса. Но заметьте, что никакие транзакции при этом не будут потеряны, так как прерываться будут не выполняющиеся транзакции, а только лишь простаивающие сеансы.

По умолчанию: 0

`server_lifetime`

pgbouncer будет закрывать неиспользуемое серверное подключение, существующее дольше заданного времени (в секундах). Ноль означает, что соединение будет использоваться только один раз, а затем будет закрываться.

По умолчанию: 3600.0

`server_idle_timeout`

Если подключение к серверу простаивает дольше заданного времени (в секундах), оно будет сброшено. При значении 0 тайм-аут отключается.

По умолчанию: 600.0

`server_connect_timeout`

Если инициализация подключения и вход на сервер не завершается за указанное время (в секундах), соединение будет закрыто.

По умолчанию: 15.0

`server_login_retry`

При ошибке входа на сервер, из-за сбоя в connect() или при аутентификации, pgbouncer будет ждать заданное время (в секундах).

По умолчанию: 15.0

`client_login_timeout`

Если клиент подключается, но не может пройти аутентификацию за указанное время (в секундах), он будет отключён. Требуется в основном, чтобы «мёртвые» подключения не задерживали операцию `SUSPEND` и, как следствие, перезагрузку «на лету».

По умолчанию: 60.0

`autodb_idle_timeout`

Если создаваемые автоматически (через «*») пулы баз данных не используются заданное время (в секундах), они освобождаются. Минусом этого является то, что их статистика также сбрасывается.

По умолчанию: 3600.0

`dns_max_ttl`

Время кеширования результатов поиска в DNS (в секундах). Если при поиске в DNS возвращаются разные ответы, pgbouncer будет использовать их по очереди. Действительное значение DNS TTL игнорируется.

По умолчанию: 15.0

`dns_nxdomain_ttl`

Время кеширования ошибок DNS и результатов NXDOMAIN (в секундах).

По умолчанию: 15.0

`dns_zone_check_period`

Интервал проверки серийного номера зоны.

pgbouncer может собрать список зон DNS из имён узлов (всё после первой точки) и затем периодически проверять, не изменился ли серийный номер зоны. Если он меняется, то все имена узлов, относящиеся к зоне, разрешаются заново. Если для какого-либо узла получается другой IP-адрес, его подключения признаются недействительными.

По умолчанию: 0.0 (отключено)

`resolv_conf`

Расположение пользовательского файла `resolv.conf`, в котором можно указать адреса серверов DNS а также другие параметры разрешения имён, не зависящие от глобальной конфигурации операционной системы.

Обработку этого файла выполняет используемая библиотека DNS, а не pgbouncer, поэтому, чтобы ознакомиться с его синтаксисом и указаниями, обратитесь к документации этой библиотеки.

По умолчанию: не задан (используются параметры операционной системы)

Параметры TLS

`client_tls_sslmode`

Режим TLS, выбираемый для подключений клиентов. По умолчанию подключения с использованием TLS запрещены. Когда этот режим включается, необходимо также настроить в `client_tls_key_file` и `client_tls_cert_file` ключ и сертификат, которые будет использовать pgbouncer.

`disable`

Обычный TCP. Если клиент запрашивает TLS, его запрос игнорируется. Это режим по умолчанию.

`allow`

Если клиент запрашивает TLS, он используется. В противном случае используется обычный протокол TCP. Если клиент предоставляет свой сертификат, он не проверяется.

`prefer`

То же, что и `allow`.

`require`

Клиент должен использовать TLS. В противном случае соединение клиента сбрасывается. Если клиент предоставляет свой сертификат, он не проверяется.

`verify-ca`

Клиент должен использовать TLS с годным клиентским сертификатом.

`verify-full`

То же, что и `verify-ca`.

`client_tls_key_file`

Закрытый ключ pgbouncer, применяемый для шифрования клиентских подключений.

По умолчанию: не задано.

`client_tls_cert_file`

Сертификат частного ключа. Клиенты могут проверить его.

По умолчанию: не задано.

`client_tls_ca_file`

Файл с корневым сертификатом, по которому будут проверяться клиентские сертификаты.

По умолчанию: не задано.

`client_tls_protocols`

Определяет, какие версии протокола TLS разрешены. Допустимые значения: `tlsv1.0`, `tlsv1.1`, `tlsv1.2`, `tlsv1.3`. Краткие обозначения: `all` (tlsv1.0,tlsv1.1,tlsv1.2,tlsv1.3), `secure` (tlsv1.2,tlsv1.3), `legacy` (равнозначно all).

По умолчанию: `secure`

`client_tls_ciphers`

Определяет допустимые шифры TLS, в синтаксисе OpenSSL. Допустимые краткие значения: `default`/`secure`, `compat`/`legacy`, `insecure`/`all`, `normal`, `fast`.

Распространяется только на соединения, использующие TLS версии 1.2 и ниже. В настоящее время нет параметра, который бы управлял выбором шифров при использовании TLS версии 1.3.

По умолчанию: `fast`

`client_tls_ecdhcurve`

Имя эллиптической кривой, применяемой при обмене ключами ECDH.

Допустимые значения: `none` (DH отключён), `auto` (256-битный ECDH), имя кривой.

По умолчанию: `auto`

`client_tls_dheparams`

Тип обмена ключами DHE.

Допустимые значения: `none` (DH отключён), `auto` (2048-битный DH), `legacy` (1024-битный DH).

По умолчанию: `auto`

`server_tls_sslmode`

Режим TLS для подключений к серверам Postgres Pro. По умолчанию подключения не используют TLS.

`disable`

Обычный TCP. TLS даже не запрашивается у сервера. Это режим по умолчанию.

`prefer`

Сначала всегда запрашивается TLS-подключение к Postgres Pro, а в случае отказа происходит переключение на простой TCP. Сертификат сервера не проверяется.

`require`

Подключение обязательно должно устанавливаться через TLS. Если сервер не принимает его, простой TCP использоваться не будет. Сертификат сервера не проверяется.

`verify-ca`

Подключение должно устанавливаться через TLS, а сертификат сервера должен быть действительным согласно файлу `server_tls_ca_file`. Имя узла сервера по сертификату не проверяется.

`verify-full`

Подключение должно устанавливаться через TLS, а сертификат сервера должен быть действительным согласно файлу `server_tls_ca_file`. Имя узла должно соответствовать указанному в сертификате.

`server_tls_ca_file`

Файл с корневым сертификатом, по которому будут проверяться сертификаты сервера Postgres Pro.

По умолчанию: не задано.

`server_tls_key_file`

Закрытый ключ pgbouncer, с которым он будет аутентифицироваться на сервере Postgres Pro.

По умолчанию: не задано.

`server_tls_cert_file`

Сертификат закрытого ключа. Сервер Postgres Pro может проверять его.

По умолчанию: не задано.

`server_tls_protocols`

Определяет, какие версии протокола TLS разрешены. Допустимые значения: `tlsv1.0`, `tlsv1.1`, `tlsv1.2`, `tlsv1.3`. Краткие обозначения: `all` (tlsv1.0,tlsv1.1,tlsv1.2,tlsv1.3), `secure` (tlsv1.2,tlsv1.3), `legacy` (равнозначно all).

По умолчанию: `secure`

`server_tls_ciphers`

Определяет допустимые шифры TLS, в синтаксисе OpenSSL. Допустимые краткие значения: `default`/`secure`, `compat`/`legacy`, `insecure`/`all`, `normal`, `fast`.

Распространяется только на соединения, использующие TLS версии 1.2 и ниже. В настоящее время нет параметра, который бы управлял выбором шифров при использовании TLS версии 1.3.

По умолчанию: `fast`

Опасные тайм-ауты

Установка следующих тайм-аутов может приводить к неожиданным ошибкам.

`query_timeout`

Запросы, выполняющиеся дольше этого времени (в секундах), будут отменяться. Его значение следует выбирать лишь немногим меньшим параметра `statement_timeout` на сервере, чтобы это происходило только при проблемах в сети.

По умолчанию: 0.0 (отключено)

`query_wait_timeout`

Максимальное время, которое могут ожидать выполнения запросы (в секундах). Если запрос не назначается серверу за это время, клиент отключается. Это применяется для предотвращения захватывания подключений «зависшими» серверами.

Это также помогает в ситуациях, когда сервер отключён или база данных не принимает подключения по какой-либо причине. Если этот тайм-аут отключить, клиенты будут стоять в очереди неограниченно долго.

По умолчанию: 120

`client_idle_timeout`

Клиентские подключения, простаивающие дольше этого времени (в секундах), закрываются. Это значение должно быть больше тайм-аута подключения, установленного на стороне клиента, и применяется оно только для решения проблем с сетью.

По умолчанию: 0.0 (отключено)

`idle_transaction_timeout`

Если клиент «простаивает в транзакции» дольше этого времени (в секундах), он будет отключён.

По умолчанию: 0.0 (отключено)

`suspend_timeout`

Сколько секунд ждать сброса буфера при выполнении `SUSPEND` или перезагрузки (`-R`). Если сброс не завершился, подключение сбрасывается.

По умолчанию: 10

Низкоуровневые параметры сети

`pkt_buf`

Размер внутреннего буфера для пакетов. Влияет на размер отправляемых TCP-пакетов и общее использование памяти. Собственно пакеты libpq могут быть больше этого буфера, так что нет необходимости делать его большим.

По умолчанию: 4096

`max_packet_size`

Максимальный размер пакетов Postgres Pro, который сможет пропустить через себя pgbouncer. Один пакет представляет либо один запрос, либо строку из набора результатов. Размер всего набора результатов может быть больше.

По умолчанию: 2147483647

`listen_backlog`

Значение параметра очереди для `listen()`. Определяет, сколько неотвеченных запросов на подключение будет находиться в очереди. Когда очередь заполнена, следующие новые подключения будут сбрасываться.

По умолчанию: 128

`sbuf_loopcnt`

Устанавливает, сколько циклов должны обрабатываться данные для одного подключения, после чего нужно переходить к другим. Без этого ограничения одно подключение с большим набором результатом может занять pgbouncer на долгое время. В одном цикле обрабатываются данные размером `pkt_buf` байт. Ноль убирает ограничение.

По умолчанию: 5

`so_reuseport`

Определяет, будет ли для принимающих TCP-сокетов устанавливаться параметр `SO_REUSEPORT`. В некоторых операционных системах данный параметр позволяет запускать несколько экземпляров pgbouncer на одном компьютере, с одним и тем же принимающим портом, при этом соединения автоматически распределяются ядром. Таким образом pgbouncer может

использовать больше ядер процессора. (pgbouncer — однопоточное приложение и использует одно ядро для каждого экземпляра.)

Данный параметр имеет желаемый эффект в Linux. В системах, которые вообще не поддерживают указанный параметр сокета, включение этого параметра приведёт к ошибке.

Для всех экземпляров pgbouncer на одном компьютере должны задаваться разные значения как минимум для `unix_socket_dir` и `pidfile`, а также `logfile`, если он используется. Также обратите внимание, что при использовании этого параметра вы больше не сможете подключаться к конкретному экземпляру pgbouncer через TCP/IP, что может затруднить мониторинг и сбор метрик.

По умолчанию: 0

`tcp_defer_accept`

Подробнее об этом и других параметрах TCP можно узнать в руководстве `man 7 tcp`.

По умолчанию: 45 в Linux, в других системах — 0

`tcp_socket_buffer`

По умолчанию: не задано.

`tcp_keepalive`

Включает базовый опрос активности со стандартными параметрами ОС.

В Linux системные параметры по умолчанию: **tcp_keepidle=7200**, **tcp_keepintvl=75**, **tcp_keepcnt=9**. Вероятно, они имеют близкие значения и в других ОС.

По умолчанию: 1

`tcp_keepcnt`

По умолчанию: не задано.

`tcp_keepidle`

По умолчанию: не задано.

`tcp_keepintvl`

По умолчанию: не задано.

`tcp_user_timeout`

Задаёт параметр сокета `TCP_USER_TIMEOUT`. Он определяет максимальное количество времени в миллисекундах, в течение которого передаваемые данные могут оставаться неподтверждёнными, прежде чем соединение TCP будет принудительно закрыто. Если установлено значение 0, используются параметры операционной системы по умолчанию.

В настоящее время поддерживается только в Linux.

По умолчанию: 0

Раздел [databases]

Этот раздел содержит пары ключ=значение, где в качестве ключа принимается имя базы данных, а в качестве значения — строка подключения для libpq в виде пар ключ=значение. Так как сама libpq не используется, в этой строке можно задать не все свойства, которые понимает libpq (service=, .pgpass).

Имя базы данных может содержать символы `_0-9A-Za-z` без кавычек. Имена, содержащие другие символы, должны заключаться в двойные кавычки по правилам для идентификаторов SQL (две кавычки ("") воспринимаются внутри строки как одна).

Имя базы данных pgbouncer зарезервировано для административной консоли и не может использоваться здесь в качестве ключа.

«*» воспринимается как имя всех остальных баз: если точного соответствия имени для запрошенной базы данных не находится, в качестве строки подключения выбирается данное значение. Например, если есть следующая запись (и нет других переопределяющих записей):

```
* = host=foo
```

В этом случае подключение к pgbouncer с указанием базы данных bar будет работать так, как если бы существовала следующая запись (в качестве dbname по умолчанию используется имя базы, полученное от клиента):

```
bar = host=foo dbname=bar
```

Такие автоматически создаваемые записи баз данных очищаются, если они простаивают больше времени, задаваемого параметром autodb_idle_timeout.

dbname

Имя целевой базы данных.

По умолчанию: имя базы данных, полученное от клиента

host

Имя или IP-адрес компьютера, к которому нужно подключиться. Имена компьютеров разрешаются в момент подключения, и результат кеширует в течение времени, заданного параметром dns_max_ttl. Если результат разрешения имени меняется, существующие подключения к серверу автоматически закрываются при их освобождении (в соответствии с режимом пула), и изменение немедленно отражается на новых подключениях. Если DNS возвращает несколько записей, они используются по очереди.

Если значение начинается с /, используется сокет Unix в пространстве имён файловой системы. Если значение начинается с @, используется сокет Unix в абстрактном пространстве имён.

По умолчанию: не задан, что подразумевает использование сокетов Unix

port

По умолчанию: 5432

user

Если задано user=, все подключения к целевой базе данных будут выполняться с заданным именем пользователя, что означает, что для этой базы данных будет всего один пул.

В противном случае pgbouncer подключается к целевой базе данных с именем пользователя, переданным клиентом, что означает, что для каждого пользователя будет отдельный пул.

password

Если пароль здесь не задаётся, будет использован пароль из auth_file или auth_query.

auth_user

Переопределяет глобальную переменную auth_user, если она задана.

pool_size

Задаёт максимальный размер пулов для этой базы данных. Если не задано, применяется значение default_pool_size.

`min_pool_size`

Задаёт минимальный размер пулов для этой базы данных. Если не задано, применяется глобальное значение `min_pool_size`.

`reserve_pool`

Задаёт число дополнительных подключений для этой базы данных. Если не задано, применяется значение `reserve_pool_size`.

`connect_query`

Запрос, который будет выполняться сразу после установления соединения, но до того, как его смогут использовать какие-либо клиенты. Если при запросе возникают ошибки, они только фиксируются в журнале, другой реакции не следует.

`pool_mode`

Задаёт режим пула для данной базы данных. Если этот параметр не задаётся, применяется значение `pool_mode` по умолчанию.

`max_db_connections`

Задаёт максимум подключений для базы данных (то есть, используя все пулы этой базы данных, нельзя будет установить больше этого числа подключений к серверу).

`client_encoding`

Запрашивает у сервера использование указанной клиентской кодировки (`client_encoding`).

`datestyle`

Запрашивает у сервера использование указанного стиля даты (`datestyle`).

`timezone`

Запрашивает у сервера использование указанного часового пояса (`timezone`).

Раздел [users]

Этот раздел содержит пары ключ=значение, где в качестве ключа принимается имя пользователя, а в качестве значения — переопределяемые для него параметры конфигурации в виде пар ключ=значение (в формате строк подключения libpq). Таким образом переопределить можно лишь немногие параметры.

`pool_mode`

Задаёт режим пула для всех подключений данного пользователя. Если этот параметр не задаётся, применяется значение `pool_mode` по умолчанию или заданное для базы данных.

`max_user_connections`

Задаёт максимум подключений для пользователя (то есть, используя все пулы, нельзя будет установить больше этого числа подключений к серверу).

Директива включения

Файл конфигурации pgbouncer может содержать директивы включения, которые указывают, что нужно прочитать и обработать дополнительный файл конфигурации. Это позволяет разделить файл конфигурации на физически отдельные части. Директивы включения выглядят примерно так:

`%include` *`имя_файла`*

Если файл задаётся не абсолютным путём, его путь воспринимается относительно текущего рабочего каталога.

Формат файла аутентификации

В этом разделе описывается формат файла, задаваемого параметром auth_file. Это текстовый файл следующего вида:

```
"username1" "password" ...
"username2" "md5abcdef012342345" ...
"username2" "SCRAM-SHA-256$число_итераций:соль$сохранённый_ключ:ключ_сервера"
```

В строке должно быть минимум два поля, заключённых в двойные кавычки. В первом поле задаётся имя пользователя, а во втором — пароль, либо открытым текстом, либо защищённый MD5 или SCRAM. Остальное содержимое строки pgbouncer игнорирует. Кавычки в этой строке можно записать, продублировав их.

Принятый в Postgres Pro формат пароля, защищённого MD5:

```
"md5" + md5(password + username)
```

Таким образом, для пользователя admin с паролем 1234 защищённый MD5 пароль будет следующим: md545f2603610af569b6155c45067268c6b.

Принятый в Postgres Pro формат пароля, защищённого SCRAM:

```
SCRAM-SHA-256$число_итераций:соль$сохранённый_ключ:ключ_сервера
```

Пароли или шифры, записанные в файле аутентификации, имеют два предназначения. Во-первых, если для клиента настроен метод аутентификации по паролю, то пароль пользователя сверяется с записанным в этом файле. Во-вторых, если сервер БД требует аутентификации по паролю (и пароль не задан явно в строке подключения), пароль для исходящего подключения к этому серверу также будет взят из этого файла. Второй вариант использования возможен при условии, что пароль записан в открытом виде или в виде MD5-хеша. Пароли в формате SCRAM могут использоваться для входа на сервер, только если клиент проходит аутентификацию по протоколу SCRAM, в определении базы pgbouncer не задаётся имя пользователя, а шифры SCRAM на сервере Postgres Pro и в pgbouncer идентичны (совпадает не только пароль, но и соль, а также количество итераций). Эти условия объясняются присущим механизму SCRAM свойством — из шифра SCRAM нельзя извлечь данные, требующиеся для аутентификации.

Файл аутентификации можно составить вручную, но также можно сгенерировать его из какого-то другого списка пользователей и паролей. В качестве примера скрипта, который генерирует файл с данными аутентификации из таблицы pg_shadow, может быть полезен ./etc/mkauth.py.

Вы также можете использовать auth_query вместо auth_file и таким образом обойтись без отдельного файла аутентификации.

Формат файла HBA

Расположение файла HBA определяется параметром auth_hba_file. Он используется, только если для auth_type задано значение hba.

Соответствует формату Postgres Pro pg_hba.conf, описанному в Разделе 19.1.

- Поддерживаемые типы записей: local, host, hostssl, hostnossl.
- В поле базы данных поддерживаются варианты: all, sameuser, @*файл*, несколько имён. Не поддерживаются: replication, samerole, samegroup.
- В поле имени пользователя поддерживаются варианты: all, @*файл*, несколько имён. Не поддерживается: +groupname.
- В поле адреса поддерживается: IPv4, IPv6. Не поддерживаются: имена DNS, префиксы доменов.
- В поле метода аутентификации поддерживаются только те, которые поддерживает pgbouncer в auth_type, за исключением any и pam, которые работают только глобально. Сопоставления имён пользователей (map=) не поддерживаются.

Пример

Минимальная конфигурация:

```
[databases]
template1 = host=localhost dbname=template1 auth_user=someuser

[pgbouncer]
pool_mode = session
listen_port = 6432
listen_addr = localhost
auth_type = md5
auth_file = users.txt
logfile = pgbouncer.log
pidfile = pgbouncer.pid
admin_users = someuser
stats_users = stat_collector
```

Параметры базы данных по умолчанию:

```
[databases]

; подключение к foodb через сокет Unix
foodb =

; перенаправление bardb в базу bazdb на локальном компьютере
bardb = host=localhost dbname=bazdb

; обращение к целевой базе данных будет производить один пользователь
forcedb = host=localhost port=300 user=baz password=foo client_encoding=UNICODE
 datestyle=ISO
```

Пример безопасной функции для `auth_query`:

```
CREATE OR REPLACE FUNCTION pgbouncer.user_lookup(in i_username text, out uname text,
 out phash text)
RETURNS record AS $$
BEGIN
    SELECT usename, passwd FROM pg_catalog.pg_shadow
    WHERE usename = i_username INTO uname, phash;
    RETURN;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;
REVOKE ALL ON FUNCTION pgbouncer.user_lookup(text) FROM public, pgbouncer;
GRANT EXECUTE ON FUNCTION pgbouncer.user_lookup(text) TO pgbouncer;
```

pg_filedump

pg_filedump — отображает отформатированное содержимое файлов кучи, индексов или управляющего файла Postgres Pro

Синтаксис

`pg_filedump` [*параметр*...] [*файл*]

Описание

pg_filedump — утилита для представления управляющего файла и файлов кучи/индекса Postgres Pro в удобочитаемом виде. Вы можете форматировать/выводить содержимое файлов несколькими способами, описанными в разделе Параметры, а также выводить данные непосредственно в двоичном виде. Тип файла (куча/индекс) обычно определяется автоматически по содержимому блоков внутри файла. Однако чтобы прочитать содержимое файла `pg_control`, необходимо использовать параметр `-c`. По умолчанию весь файл обрабатывается с размером блока, указанным в блоке 0, и отображаются относительные адреса блоков. Значения по умолчанию можно изменить с помощью параметров времени выполнения. Некоторые параметры могут показаться странными, но они необходимы. Например, размер блока может потребоваться задать, если заголовок блока 0 был повреждён.

Установка

pg_filedump поставляется вместе с Postgres Pro в виде отдельного пакета `pg-filedump-std-10` (подробные инструкции по установке приведены в Главе 16).

Параметры

По умолчанию выводится содержимое всего файла, с относительными адресами и размером блока, полученным из блока 0.

Для файлов кучи и файлов индекса действуют следующие параметры:

`-a`

Отображать абсолютные адреса. Содержимое заголовка блока всегда адресуется относительно блока.

`-b`

Выводить двоичные образы блоков в заданном диапазоне. При этом все остальные параметры форматирования не действуют.

`-d`

Показать отформатированный вывод байтов блока. При этом все остальные параметры форматирования не действуют.

`-D` *список_атрибутов*

Декодировать кортежи, используя заданный список типов, разделённых запятыми. Список поддерживаемых типов:

`bigint`
`bigserial`
`bool`
`char`
`charN` — char(n)
`date`
`float`
`float4`

`float8`
`int`
`json`
`macaddr`
`name`
`numeric`
`oid`
`real`
`serial`
`smallint`
`smallserial`
`text`
`time`
`timestamp`
`timetz`
`uuid`
`varchar`
`varcharN` — varchar(n)
`xid`
`xml`
`~` — игнорировать все последующие атрибуты в кортеже.

`-f`

Показать отформатированный вывод байтов блока вместе с интерпретацией.

`-h`

Показать справку.

`-i`

Выводить подробную информацию об интерпретируемых элементах.

`-k`

Проверять контрольные суммы блоков.

`-o`

Не выводить старые значения.

`-R` *`начальный_блок`* [*`конечный_блок`*]

Показать только заданные диапазоны блоков в файле. Блоки нумеруются, начиная с 0. Параметры *`начальный_блок`* и *`конечный_блок`* определяют первый и последний блок диапазона, соответственно. Если *`конечный_блок`* опускается, а задаётся только *`начальный_блок`*, будет обработан только один этот блок.

`-s` *`размер_сегмента`*

Задать размер сегмента равным значению *`размер_сегмента`*.

`-t`

Обрабатывать файлы TOAST.

`-v`

Вывести дополнительную информацию об отношениях TOAST.

`-n` *`номер_сегмента`*

Задать номер сегмента равным значению *`номер_сегмента`*.

`-S` *`размер_блока`*

Задать размер блока равным значению *`размер_блока`*.

`-x`

Форматировать элементы блока как элементы индекса.

`-y`

Форматировать элементы блока как элементы кучи.

Для управляющих файлов действуют следующие параметры:

`-c`

Интерпретировать указанный файл как управляющий файл.

`-f`

Показать отформатированный вывод байтов содержимого вместе с интерпретацией.

`-S` *`размер_блока`*

Задать размер блока равным значению *`размер_блока`*.

Дополнительные параметры:

`-m`

Интерпретировать файл как файл `pg_filenode.map` и вывести его содержимое. Все остальные параметры будут проигнорированы.

В большинстве случаев рекомендуется использовать параметры `-i` и `-f`, чтобы получить наиболее информативный вывод.

Автор

Патрик Макдональд <patrickm@redhat.com>

pgpro_controldata

pgpro_controldata — вывести управляющую информацию кластера БД PostgreSQL/Postgres Pro и параметры совместимости кластера и/или сервера.

Синтаксис

`pgpro_controldata` [*параметр*...]

Описание

`pgpro_controldata` выводит управляющую информацию, в частности, версию каталога, записываемую программой `initdb` при инициализации любого кластера PostgreSQL/Postgres Pro. Данная утилита также выводит сведения о журнале предзаписи, включая данные контрольной точки. Эта информация относится ко всему кластеру, а не к отдельной базе данных.

`pgpro_controldata` также помогает проверить совместимость между серверами и кластерами БД PostgreSQL/Postgres Pro. Эта программа может вывести параметры, влияющие на совместимость, и определить, совместим ли сервер с кластером, сравнивая эти параметры.

Параметры

`pgpro_controldata` принимает в командной строке параметры, описанные далее. В случае запуска без параметров `pgpro_controldata` просто выводит управляющую информацию так, как это делает pg_controldata. Заметьте, что аргументы командной строки, связанные с совместимостью, `-P` и `-S`, работают в паре как один аргумент `-C`.

Параметры общего назначения

`-B`
`--bindir`

Задаёт каталог с исполняемыми файлами PostgreSQL/Postgres Pro; требуется для получения параметров совместимости сервера.

`-D` *каталог_данных*
`--pgdata=`*каталог_данных*

Указывает размещение конфигурационных файлов кластера. Если этот ключ опущен, используется значение переменной окружения `PGDATA`.

`-V`
`--version`

Показать версию pgpro_controldata, а затем завершиться.

`-?`
`--help`

Показать справку по аргументам командной строки pgpro_controldata, а затем завершиться.

Параметры, связанные с совместимостью

`-C`
`--compatibility-check`

Вывести все параметры целевого кластера, от которых зависит совместимость выбранного сервера и кластера, и установить, совместимы ли они.

Путь к целевому каталогу данных, доступному для чтения, передаётся в аргументе `-D` или в переменной окружения `PGDATA`.

Если параметр `-B` опущен, подразумевается текущий сервер.

Эта проверка будет полноценной, только если данные кластера и сервер рассчитаны на одну архитектуру процессора и одинаковый порядок байтов.

`-P`
`--cluster-compatibility-params`

Вывести все параметры целевого кластера, которые могут повлиять на совместимость.

Путь к целевому каталогу данных, доступному для чтения, передаётся в аргументе `-D` или в переменной окружения `PGDATA`.

Эта проверка будет полноценной, только если данные кластера и сервер рассчитаны на одну архитектуру процессора и одинаковый порядок байтов.

`-S`
`--server-compatibility-params`

Вывести все параметры указанного или текущего сервера, которые могут повлиять на совместимость.

Если параметр `-B` опущен, подразумевается текущий сервер.

Переменные окружения

`PGDATA`

Каталог размещения данных кластера по умолчанию

`PG_COLOR`

Выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

См. также

pg_controldata

pg_standby

pg_standby — поддерживает создание сервера тёплого резерва Postgres Pro

Синтаксис

`pg_standby` [*параметр*...] *расположение_архива* *следующий_файл_wal* *каталог_wal* [*файл_перезапуска_wal*]

Описание

Программа pg_standby поддерживает создание сервера в режиме «тёплого резерва». Она предназначена как для непосредственного применения в производственной среде, так и для использования в качестве настраиваемой заготовки, когда требуются специальные модификации.

pg_standby ожидает выполнения команды `restore_command`, которая, в свою очередь, нужна для перехода от стандартного восстановления архива к режиму тёплого резерва. Для этого также требуется другая настройка, которая описывается в основном руководстве сервера (см. Раздел 25.2).

Чтобы настроить резервный сервер на использование pg_standby, поместите эту строку в файл конфигурации `recovery.conf`:

```
restore_command = 'pg_standby каталог_архива %f %p %r'
```

Здесь *каталог_архива* — каталог, из которого должны восстанавливаться сегменты WAL.

Если указывается *файл_перезапуска_wal*, обычно с помощью макроса `%r`, тогда все файлы WAL, предшествующие указанному, будут удалены из каталога *расположение_архива*. Это позволяет сократить число сохраняемых файлов без потери возможности восстановления при перезапуске. Такой вариант использования уместен, когда *расположение_архива* указывает на область рабочих файлов конкретного резервного сервера, но *не* когда *расположение_архива* — каталог с архивом WAL для долговременного хранения.

pg_standby рассчитывает на то, что *расположение_архива* доступно для чтения пользователю, владеющему серверным процессом. Если указывается *файл_перезапуска_wal* (или `-k`), каталог *расположение_архива* должен быть также доступен для записи.

При отказе ведущего сервера переключение на сервер «тёплого резерва» возможно двумя способами:

Умное переключение

> При умном переключении сервер включается в работу, применив изменения из всех файлов WAL, имеющихся в архиве. В результате никакие данные не теряются, даже если данный резервный сервер отстал, но если применить нужно большое количество изменений WAL, подготовка к работе может быть длительной. Чтобы вызвать умное переключение, создайте файл-триггер, содержащий слово `smart`, либо просто пустой файл.

Быстрое переключение

> При быстром переключении сервер включается в работу немедленно. Все ещё не применённые файлы WAL в архиве будут игнорироваться, и все транзакции в этих файлах будут потеряны. Чтобы вызвать быстрое переключение, создайте файл-триггер и запишите в него слово `fast`. Программу pg_standby можно также настроить так, чтобы быстрое переключение происходило автоматически, если за определённое время не появляется новый файл WAL.

Параметры

pg_standby принимает следующие аргументы командной строки:

`-c`

Применять для восстановления файлов WAL из архива команду `cp` или `copy`. На данный момент поддерживается только это поведение, так что этот параметр бесполезен.

`-d`

Выводить подробные отладочные сообщения в `stderr`.

`-k`

Удалить файлы из каталога *`расположение_архива`*, чтобы в нём осталось не больше заданного числа файлов WAL, предшествующих текущему. Ноль (по умолчанию) означает, что не нужно удалять никакие файлы из каталога *`расположение_архива`*. Этот параметр будет просто игнорироваться, если указан *`файл_перезапуска_wal`*, так как этот метод более точно определяет правильную точку отсечения архива. Этот параметр считается *устаревшим* с PostgreSQL 8.3; надёжнее и эффективнее использовать параметр *`файл_перезапуска_wal`*. При слишком маленьком значении данного параметра могут быть удалены файлы, требующиеся для перезапуска резервного сервера, тогда как при слишком большом будет неэффективно расходоваться место в архиве.

`-r` *`макс_повторов`*

Устанавливает, сколько раз максимум нужно повторять команду copy в случае ошибки (по умолчанию 3). После каждой ошибки программа приостанавливается на *`время_задержки`* * *`число_повторов`*, так что время ожидания постепенно увеличивается. По умолчанию она ждёт 5, 10, затем 15 секунд, и только потом сообщает резервному серверу об ошибке. Это событие будет воспринято как завершение восстановления, и в результате резервный сервер полностью включится в работу.

`-s` *`время_задержки`*

Задаёт количество секунд (до 60, по умолчанию 5) для паузы между проверками наличия файла WAL в архиве. Значение по умолчанию не обязательно наилучшее; за подробностями обратитесь к Разделу 25.2.

`-t` *`файл_триггер`*

Указывает файл-триггер, при появлении которого должна начаться отработка отказа. Имя этого файла рекомендуется выбирать по определённой схеме, позволяющей однозначно понять, для какого сервера вызывается отработка отказа, когда таких серверов в одной системе несколько; например, `/tmp/pgsql.trigger.5432`.

`-V`
`--version`

Вывести версию pg_standby и завершиться.

`-w` *`макс_время_ожидания`*

Задаёт максимальное время ожидания (в секундах) следующего файла WAL, по истечении которого будет произведено быстрое переключение. При нуле (значении по умолчанию) ожидание бесконечно. Значение по умолчанию не обязательно наилучшее; за подробностями обратитесь к Разделу 25.2.

`-?`
`--help`

Вывести справку об аргументах командной строки pg_standby и завершиться.

Замечания

Программа pg_standby предназначена для работы с PostgreSQL 8.2 и новее.

С PostgreSQL, начиная с 8.3, можно использовать макрос `%r`, который позволяет pg_standby узнать, какой последний файл нужно сохранять. Для PostgreSQL 8.2, если требуется очищать архив, нужно применять параметр `-k`. Этот параметр сохранился и после 8.3, но теперь он считается устаревшим.

PostgreSQL, начиная с 8.4, поддерживает параметр `recovery_end_command`. В нём можно задать команду, удаляющую файл-триггер во избежание ошибок.

Программа pg_standby написана на C; её исходный код легко поддаётся модификации (он содержит секции, предназначенные для изменения при надобности)

Примеры

В системах Linux или Unix можно использовать команды:

```
archive_command = 'cp %p .../archive/%f'

restore_command = 'pg_standby -d -s 2 -t /tmp/pgsql.trigger.5442 .../archive %f %p %r
 2>>standby.log'

recovery_end_command = 'rm -f /tmp/pgsql.trigger.5442'
```

Предполагается, что каталог архива физически располагается на резервном сервере, так что команда `archive_command` обращается к нему по NFS, но для резервного сервера эти файлы локальные (для этого применяется `ln`). Эти команды будут:

- выводить отладочную информацию в `standby.log`
- ждать 2 секунды между проверками появления следующего файла WAL
- прекращать ожидание, только когда появляется файл-триггер с именем `/tmp/pgsql.trigger.5442`, и выполнить переключение согласно его содержимому
- удалять файл-триггер по завершении восстановления
- удалять ставшие ненужными файлы из каталога архива

В Windows можно использовать такие команды:

```
archive_command = 'copy %p ...\\archive\\%f'

restore_command = 'pg_standby -d -s 5 -t C:\pgsql.trigger.5442 ...\archive %f %p %r
 2>>standby.log'

recovery_end_command = 'del C:\pgsql.trigger.5442'
```

Заметьте, что обратную косую черту нужно дублировать в `archive_command`, но *не* в `restore_command` или `recovery_end_command`. Эти команды будут:

- применять команду `copy` для восстановления файлов WAL из архива
- выводить отладочную информацию в `standby.log`
- ждать 5 секунд между проверками появления следующего файла WAL
- прекращать ожидание, только когда появляется файл-триггер с именем `C:\pgsql.trigger.5442`, и выполнить переключение согласно его содержимому
- удалять файл-триггер по завершении восстановления
- удалять ставшие ненужными файлы из каталога архива

Команда `copy` в Windows устанавливает окончательный размер файла до того, как файл будет окончательно скопирован, что обычно сбивает с толку pg_standby. Поэтому pg_standby ждёт *`время_задержки`* после того, как увидит подходящий размер файла. Команда `cp` из GNUWin32 устанавливает размер файла, только когда завершает копирование.

Так как в примере для Windows с обеих сторон применяется `copy`, любой или оба этих сервера могут обращаться к каталогу архива по сети.

Автор

Саймон Риггс `<simon@2ndquadrant.com>`

См. также

pg_archivecleanup

Приложение H. Внешние проекты

Postgres Pro — сложный программный проект, управлять которым довольно непросто. Поэтому мы пришли к заключению, что многие дополнения и усовершенствования Postgres Pro будет эффективнее разрабатывать отдельно от основного проекта.

H.1. Клиентские интерфейсы

В базовый дистрибутив Postgres Pro Standard включены только два клиентских интерфейса:

- libpq включён, потому что это основной интерфейс языка C и многие другие клиентские интерфейсы построены на основе него.
- ECPG включён, потому что он зависит от грамматики языка SQL на стороне сервера, и таким образом, очень чувствителен к изменениям в самом Postgres Pro.

Все остальные языковые интерфейсы разрабатываются в отдельных проектах и распространяются отдельно. Некоторые из этих проектов перечислены в Таблице H.1. Заметьте, что какие-то проекты могут выпускаться под лицензией, отличной от лицензии Postgres Pro. За дополнительной информацией о каждом языковом интерфейсе, включая условия лицензии, обратитесь к его сайту и документации.

Таблица H.1. Отдельно поддерживаемые клиентские интерфейсы

Название	Язык	Комментарии	Сайт
DBD::Pg	Perl	DBI-драйвер для Perl	*https://metacpan.org/release/DBD-Pg/*
JDBC	Java	JDBC-драйвер типа 4	*https://jdbc.postgresql.org/*
libpqxx	C++	Интерфейс C++	*https://pqxx.org/*
node-postgres	JavaScript	Драйвер Node.js	*https://node-postgres.com/*
Npgsql	.NET	Провайдер данных для .NET	*https://www.npgsql.org/*
pgtcl	Tcl		*https://github.com/flightaware/Pgtcl*
pgtclng	Tcl		*http://sourceforge.net/projects/pgtclng/*
pq	Go	Драйвер на Go для интерфейса database/sql	*https://github.com/lib/pq*
psqlODBC	ODBC	ODBC-драйвер	*https://odbc.postgresql.org/*
psycopg	Python	Интерфейс, совместимый с DB API 2.0	*https://www.psycopg.org/*

H.2. Средства администрирования

Для Postgres Pro выпускаются различные инструменты администрирования. Наиболее популярен из них *pgAdmin*, но есть и ряд других, в том числе коммерческих решений.

H.3. Процедурные языки

Базовый дистрибутив Postgres Pro включает несколько процедурных языков: PL/pgSQL, PL/Tcl, PL/Perl и PL/Python.

Кроме того, вне основного дистрибутива Postgres Pro разрабатываются и поддерживаются несколько процедурных языков. Проекты некоторых из этих языков перечислены в Таблице H.2. Заметьте, что какие-то проекты могут выпускаться под лицензией, отличной от лицензии Postgres Pro. За дополнительной информацией о каждом процедурном языке, включая условия лицензии, обратитесь к его сайту и документации.

Таблица H.2. Поддерживаемые отдельно процедурные языки

Название	Язык	Сайт
PL/Java	Java	*https://tada.github.io/pljava/*
PL/Lua	Lua	*https://github.com/pllua/pllua-ng*
PL/R	R	*https://github.com/postgres-plr/plr*
PL/sh	Оболочка Unix	*https://github.com/petere/plsh*
PL/v8	JavaScript	*https://github.com/plv8/plv8*

H.4. Расширения

Postgres Pro проектируется так, чтобы его можно было легко расширять. Как результат, расширения, загружаемые в базу данных, могут функционировать в точности так же, как и встроенные функции. Каталог `contrib/`, включённый в дерево исходного кода, содержит несколько расширений, описанных в Приложении F. Другие расширения разрабатываются независимо, как например, *PostGIS*. Postgres Pro позволяет разрабатывать независимо даже решения по репликации. Например, *Slony-I* — популярное средство репликации по схеме главный/резервный, разрабатываемое отдельно от основного проекта.

Приложение I. Настройка Postgres Pro для решений 1С

Вы можете установить и использовать Postgres Pro с решениями 1С в клиент/серверной модели.

Убедитесь, что в вашей системе установлена русская локаль (например, `ru_RU.UTF-8` в системах Linux) и она является активной локалью пользователя, создающего кластер БД. Например, в ОС Debian выполните следующие действия:

```
sudo dpkg-reconfigure locales # Выберите создаваемую локаль ru_RU.UTF-8
export LANG="ru_RU.UTF-8"
/opt/pgpro/std-10/bin/pg-setup initdb
```

Подробности можно найти в соответствующих разделах документации 1С и вашей ОС.

Для оптимальной производительности и стабильности измените следующие параметры в конфигурационном файле `postgresql.conf` сервера Postgres Pro:

1. Увеличьте максимально возможное число одновременных подключений к серверу баз данных до 1000. Решения 1С могут открывать большое количество соединений, даже если все они не нужны, так что рекомендуется разрешить на сервере не менее 500 подключений.

   ```
   max_connections = 1000
   ```

2. Чтобы временные таблицы работали корректно, измените следующие параметры:

   - Увеличьте размер буфера для временных таблиц:

     ```
     temp_buffers = 32MB
     ```

   - Увеличьте число допустимых в одной транзакции блокировок таблиц или индексов до 256:

     ```
     max_locks_per_transaction = 256
     ```

     Обычно решения 1С используют множество временных таблиц. Такие таблицы в большом количестве используются каждым обслуживающим процессом. Закрывая соединение, Postgres Pro пытается удалить все временные таблицы в одной транзакции, при этом транзакция может запрашивать множество блокировок. Если число блокировок превысит значение `max_locks_per_transaction`, транзакция прервётся и оставит за собой множество потерянных временных таблиц.

3. Включите экранирование символом обратной косой черты для всех строк и выключите предупреждение об использовании этого символа:

   ```
   standard_conforming_strings = off
   escape_string_warning = off
   ```

4. Задайте параметр `effective_cache_size` равным минимум половине объёма ОЗУ, доступного в системе. Эффективность оптимизатора запросов Postgres Pro зависит от выделенного ему объёма ОЗУ.

5. Оптимизируйте планирование запросов с помощью расширения `plantuner`:

   - Добавьте `plantuner` в переменную `shared_preload_libraries`:

     ```
     shared_preload_libraries = 'plantuner'
     ```

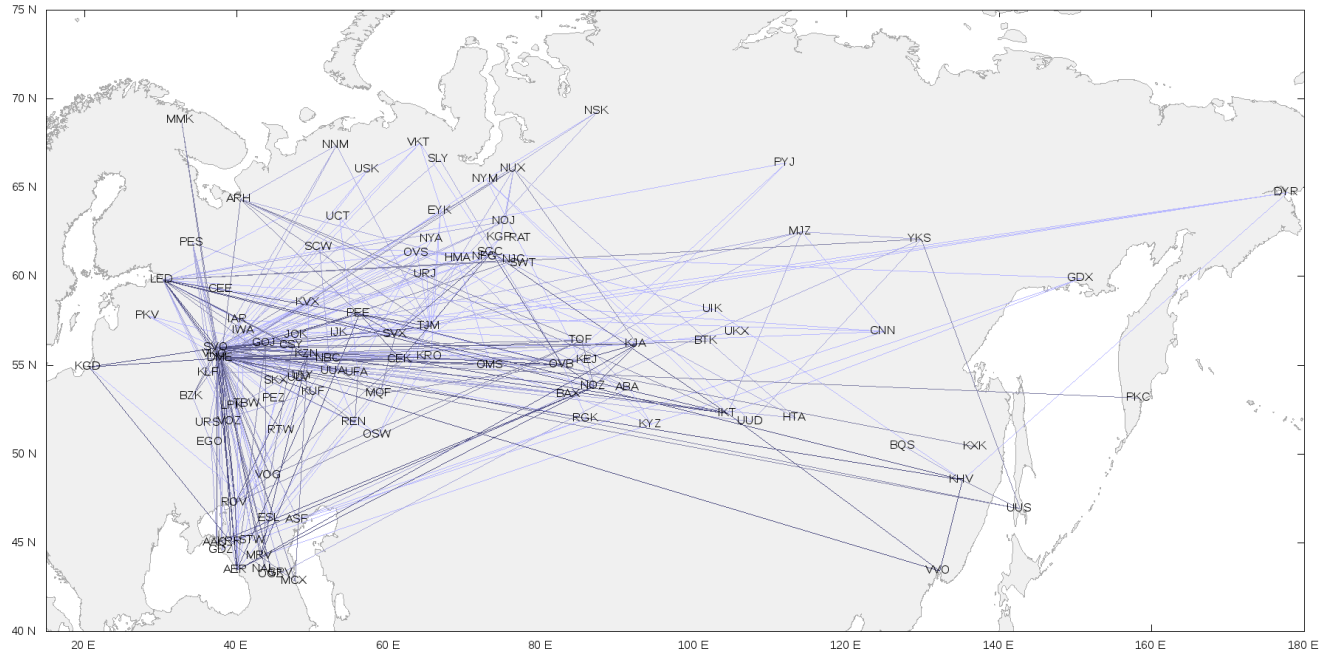
   - Настройте оптимизатор Postgres Pro для улучшенного планирования запросов с недавно созданными пустыми таблицами:

     ```
     plantuner.fix_empty_table = 'on'
     ```

Приложение J. Демонстрационная база данных «Авиаперевозки»

Представляем вам демонстрационную базу данных для Postgres Pro. В этом приложении к документации описана схема данных, состоящая из восьми таблиц и нескольких представлений. В качестве предметной области выбраны авиаперевозки по России. Базу данных можно загрузить с *нашего сайта*, см. Раздел J.1.

Рисунок J.1. Воздушное сообщение в России



База данных может использоваться, например:

- для самостоятельного изучения языка запросов SQL;
- для подготовки книг, пособий и учебных курсов по языку SQL;
- для демонстрации возможностей Postgres Pro в статьях и заметках.

При разработке демонстрационной базы данных мы преследовали несколько целей:

- схема данных должна быть достаточно простой, чтобы быть понятной без особых пояснений;
- в то же время схема данных должна быть достаточно сложной, чтобы позволять строить осмысленные запросы;
- база данных должна быть наполнена данными, напоминающими реальные, с которыми будет интересно работать.

Демонстрационная база данных распространяется под *лицензией PostgreSQL*.

Свои замечания и пожелания направляйте нам по адресу *edu@postgrespro.ru*.

J.1. Установка

Демонстрационная база данных доступна на *edu.postgrespro.ru* в трёх версиях, которые отличаются только объёмом данных:

- *demo-small.zip* (21 MB) — — данные по полётам за один месяц (размер БД около 300 МБ)
- *demo-medium.zip* (62 МБ) — данные по полётам за три месяца (размер БД около 700 МБ)

- *demo-big.zip* (232 МБ) — данные по полётам за год (размер БД около 2,5 ГБ)

Небольшая база годится для того, чтобы писать запросы, и при этом не займёт много места на диске. База большого размера позволит почувствовать, как ведут себя запросы на больших объёмах данных, и задуматься об оптимизации.

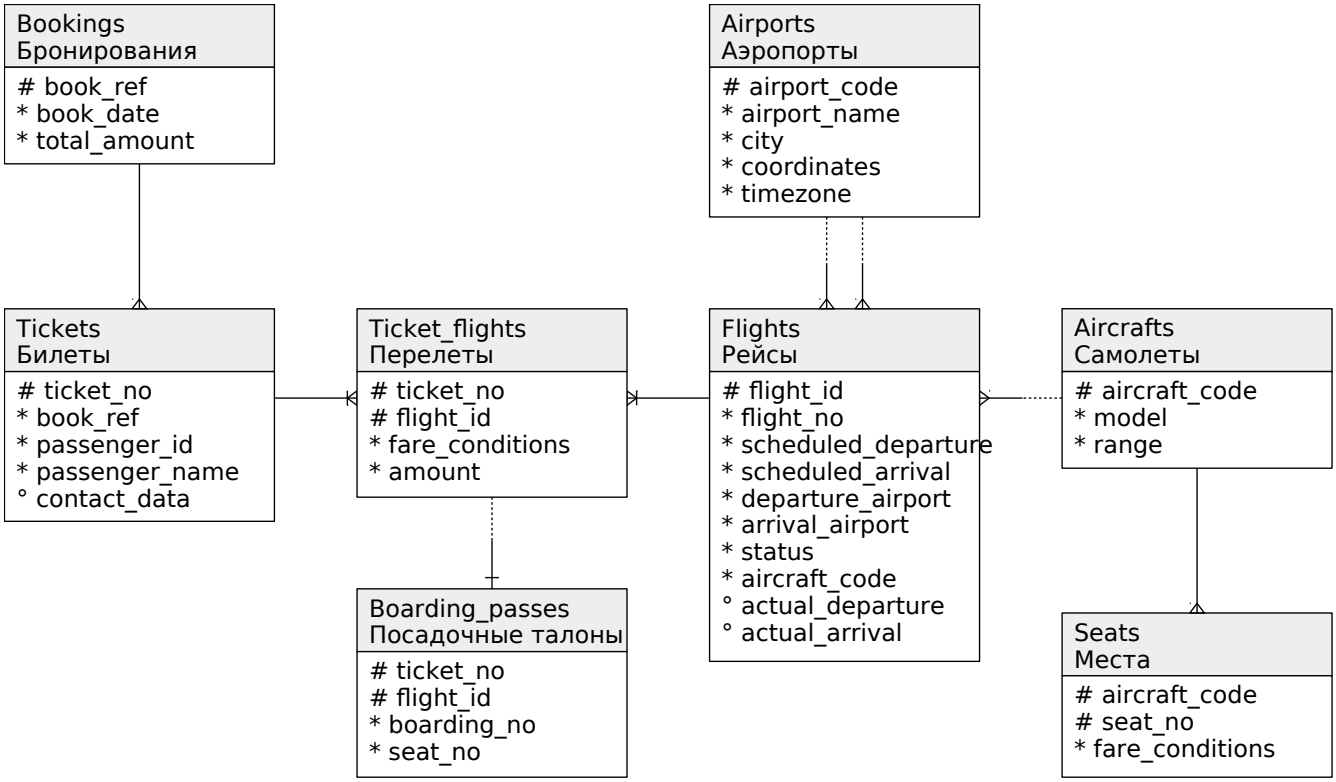
Файлы содержат SQL-скрипт, создающий базу данных `demo` и наполняющий её данными (фактически, это резервная копия, созданная утилитой pg_dump). Владельцем базы данных `demo` станет пользователь СУБД, выполнявший скрипт. Например, чтобы создать небольшую базу данных, запустите скрипт от имени пользователя `postgres` в программе psql:

```
psql -f demo_small_YYYYMMDD.sql -U postgres
```

Обратите внимание, что если база данных `demo` уже существует, она будет удалена и создана заново!

J.2. Диаграмма схемы данных

Рисунок J.2. Диаграмма схемы Bookings



J.3. Описание схемы

Основной сущностью является бронирование (`bookings`).

В одно бронирование можно включить несколько пассажиров, каждому из которых выписывается отдельный билет (`tickets`). Билет имеет уникальный номер и содержит информацию о пассажире. Как таковой пассажир не является отдельной сущностью. Как имя, так и номер документа пассажира могут меняться с течением времени, так что невозможно однозначно найти все билеты одного человека; для простоты можно считать, что все пассажиры уникальны.

Билет включает один или несколько перелетов (`ticket_flights`). Несколько перелетов могут включаться в билет в случаях, когда нет прямого рейса, соединяющего пункты отправления и назначения (полет с пересадками), либо когда билет взят «туда и обратно». В схеме данных нет жёсткого ограничения, но предполагается, что все билеты в одном бронировании имеют одинаковый набор перелетов.

Каждый рейс (flights) следует из одного аэропорта (airports) в другой. Рейсы с одним номером имеют одинаковые пункты вылета и назначения, но будут отличаться датой отправления.

При регистрации на рейс пассажиру выдаётся посадочный талон (boarding_passes), в котором указано место в самолете. Пассажир может зарегистрироваться только на тот рейс, который есть у него в билете. Комбинация рейса и места в самолете должна быть уникальной, чтобы не допустить выдачу двух посадочных талонов на одно место.

Количество мест (seats) в самолете и их распределение по классам обслуживания зависит от модели самолета (aircrafts), выполняющего рейс. Предполагается, что каждая модель самолета имеет только одну компоновку салона. Схема данных не контролирует, что места в посадочных талонах соответствуют имеющимся в самолете (такая проверка может быть сделана с использованием табличных триггеров или в приложении).

J.4. Объекты схемы

J.4.1. Список отношений

Имя	Тип	Small	Medium	Big	Описание
aircrafts	представление				Самолеты
aircrafts_data	таблица	16 kB	16 kB	16 kB	Самолеты (переводы)
airports	представление				Аэропорты
airports_data	таблица	56 kB	56 kB	56 kB	Аэропорты (переводы)
boarding_passes	таблица	31 MB	102 MB	427 MB	Посадочные талоны
bookings	таблица	13 MB	30 MB	105 MB	Бронирования
flights	таблица	3 MB	6 MB	19 MB	Рейсы
flights_v	представление				Рейсы
routes	представление				Маршруты
seats	таблица	88 kB	88 kB	88 kB	Места
ticket_flights	таблица	64 MB	145 MB	516 MB	Перелеты
tickets	таблица	47 MB	107 MB	381 MB	Билеты

J.4.2. Представление bookings.aircrafts

Каждая модель воздушного судна идентифицируется своим трёхзначным кодом (aircraft_code). Указывается также название модели (model) и максимальная дальность полета в километрах (range).

Значение поля model определяется в зависимости от выбранного языка. За подробностями обратитесь к Подразделу J.4.15.

Столбец	Тип	Модификаторы	Описание
aircraft_code	char(3)	not null	Код самолета, IATA
model	text	not null	Модель самолета
range	integer	not null	Максимальная дальность полета, км

```
Определение представления:
 SELECT ml.aircraft_code,
    ml.model ->> lang() AS model,
    ml.range
   FROM aircrafts_data ml;
```

J.4.3. Таблица bookings.aircrafts_data

Это базовая таблица для представления aircrafts. Поле model этой таблицы содержит переводы моделей самолётов на разные языки, в формате JSONB. В большинстве случаев к этой таблице не следует обращаться напрямую.

```
    Столбец    |   Тип   | Модификаторы |              Описание
---------------+---------+--------------+-----------------------------------
 aircraft_code | char(3) | not null     | Код самолета, IATA
 model         | jsonb   | not null     | Модель самолета
 range         | integer | not null     | Максимальная дальность полета, км
Индексы:
    PRIMARY KEY, btree (aircraft_code)
Ограничения-проверки:
    CHECK (range > 0)
Ссылки извне:
    TABLE "flights" FOREIGN KEY (aircraft_code)
        REFERENCES aircrafts_data(aircraft_code)
    TABLE "seats" FOREIGN KEY (aircraft_code)
        REFERENCES aircrafts_data(aircraft_code) ON DELETE CASCADE
```

J.4.4. Представление `bookings.airports`

Аэропорт идентифицируется трехбуквенным кодом (`airport_code`) и имеет своё имя (`airport_name`).

Для города не предусмотрено отдельной сущности, но введено поле с названием города (`city`), позволяющее найти аэропорты одного города. Это представление также включает координаты аэропорта (`coordinates`) и часовой пояс (`timezone`).

Значения полей `airport_name` и `city` определяются в зависимости от выбранного языка. За подробностями обратитесь к Подразделу J.4.15.

```
   Столбец    |   Тип   | Модификаторы |                Описание
--------------+---------+--------------+-----------------------------------------
 airport_code | char(3) | not null     | Код аэропорта
 airport_name | text    | not null     | Название аэропорта
 city         | text    | not null     | Город
 coordinates  | point   | not null     | Координаты аэропорта (долгота и широта)
 timezone     | text    | not null     | Часовой пояс аэропорта
Определение представления:
 SELECT ml.airport_code,
    ml.airport_name ->> lang() AS airport_name,
    ml.city ->> lang() AS city,
    ml.coordinates,
    ml.timezone
   FROM airports_data ml;
```

J.4.5. Таблица `bookings.airports_data`

Это базовая таблица для представления `airports`. Она содержит переводы значений `airport_name` и `city` на разные языки, в формате JSONB. В большинстве случаев к этой таблице не следует обращаться напрямую.

```
   Столбец    |   Тип   | Модификаторы |                Описание
--------------+---------+--------------+-----------------------------------------
 airport_code | char(3) | not null     | Код аэропорта
 airport_name | jsonb   | not null     | Название аэропорта
 city         | jsonb   | not null     | Город
 coordinates  | point   | not null     | Координаты аэропорта (долгота и широта)
 timezone     | text    | not null     | Часовой пояс аэропорта
Индексы:
    PRIMARY KEY, btree (airport_code)
```

```
Ссылки извне:
    TABLE "flights" FOREIGN KEY (arrival_airport)
        REFERENCES airports_data(airport_code)
    TABLE "flights" FOREIGN KEY (departure_airport)
        REFERENCES airports_data(airport_code)
```

J.4.6. Таблица `bookings.boarding_passes`

При регистрации на рейс, которая возможна за сутки до плановой даты отправления, пассажиру выдаётся посадочный талон. Он идентифицируется также, как и перелёт — номером билета и номером рейса.

Посадочным талонам присваиваются последовательные номера (`boarding_no`) в порядке регистрации пассажиров на рейс (этот номер будет уникальным только в пределах данного рейса). В посадочном талоне указывается номер места (`seat_no`).

```
   Столбец    |     Тип     | Модификаторы |          Описание
--------------+-------------+--------------+---------------------------
 ticket_no    | char(13)    | not null     | Номер билета
 flight_id    | integer     | not null     | Идентификатор рейса
 boarding_no | integer     | not null     | Номер посадочного талона
 seat_no      | varchar(4) | not null     | Номер места
Индексы:
    PRIMARY KEY, btree (ticket_no, flight_id)
    UNIQUE CONSTRAINT, btree (flight_id, boarding_no)
    UNIQUE CONSTRAINT, btree (flight_id, seat_no)
Ограничения внешнего ключа:
    FOREIGN KEY (ticket_no, flight_id)
        REFERENCES ticket_flights(ticket_no, flight_id)
```

J.4.7. Таблица `bookings.bookings`

Пассажир заранее (`book_date`, максимум за месяц до рейса) бронирует билет себе и, возможно, нескольким другим пассажирам. Бронирование идентифицируется номером (`book_ref`, шестизначная комбинация букв и цифр).

Поле `total_amount` хранит общую стоимость включённых в бронирование перелетов всех пассажиров.

```
   Столбец    |      Тип       | Модификаторы |          Описание
--------------+----------------+--------------+----------------------------
 book_ref     | char(6)        | not null     | Номер бронирования
 book_date    | timestamptz    | not null     | Дата бронирования
 total_amount | numeric(10,2) | not null     | Полная сумма бронирования
Индексы:
    PRIMARY KEY, btree (book_ref)
Ссылки извне:
    TABLE "tickets" FOREIGN KEY (book_ref) REFERENCES bookings(book_ref)
```

J.4.8. Таблица `bookings.flights`

Естественный ключ таблицы рейсов состоит из двух полей — номера рейса (`flight_no`) и даты отправления (`scheduled_departure`). Чтобы сделать внешние ключи на эту таблицу компактнее, в качестве первичного используется суррогатный ключ (`flight_id`).

Рейс всегда соединяет две точки — аэропорты вылета (`departure_airport`) и прибытия (`arrival_airport`). Такое понятие, как «рейс с пересадками» отсутствует: если из одного

аэропорта до другого нет прямого рейса, в билет просто включаются несколько необходимых рейсов.

У каждого рейса есть запланированные дата и время вылета (`scheduled_departure`) и прибытия (`scheduled_arrival`). Реальные время вылета (`actual_departure`) и прибытия (`actual_arrival`) могут отличаться: обычно не сильно, но иногда и на несколько часов, если рейс задержан.

Статус рейса (`status`) может принимать одно из следующих значений:

`Scheduled`

Рейс доступен для бронирования. Это происходит за месяц до плановой даты вылета; до этого запись о рейсе не существует в базе данных.

`On Time`

Рейс доступен для регистрации (за сутки до плановой даты вылета) и не задержан.

`Delayed`

Рейс доступен для регистрации (за сутки до плановой даты вылета), но задержан.

`Departed`

Самолет уже вылетел и находится в воздухе.

`Arrived`

Самолет прибыл в пункт назначения.

`Cancelled`

Рейс отменён.

```
       Столбец       |     Тип     | Модификаторы |           Описание
---------------------+-------------+--------------+------------------------------
 flight_id           | serial      | not null     | Идентификатор рейса
 flight_no           | char(6)     | not null     | Номер рейса
 scheduled_departure | timestamptz | not null     | Время вылета по расписанию
 scheduled_arrival   | timestamptz | not null     | Время прилёта по расписанию
 departure_airport   | char(3)     | not null     | Аэропорт отправления
 arrival_airport     | char(3)     | not null     | Аэропорт прибытия
 status              | varchar(20) | not null     | Статус рейса
 aircraft_code       | char(3)     | not null     | Код самолета, IATA
 actual_departure    | timestamptz |              | Фактическое время вылета
 actual_arrival      | timestamptz |              | Фактическое время прилёта
Индексы:
    PRIMARY KEY, btree (flight_id)
    UNIQUE CONSTRAINT, btree (flight_no, scheduled_departure)
Ограничения-проверки:
    CHECK (scheduled_arrival > scheduled_departure)
    CHECK ((actual_arrival IS NULL)
       OR  ((actual_departure IS NOT NULL AND actual_arrival IS NOT NULL)
            AND (actual_arrival > actual_departure)))
    CHECK (status IN ('On Time', 'Delayed', 'Departed',
                      'Arrived', 'Scheduled', 'Cancelled'))
Ограничения внешнего ключа:
    FOREIGN KEY (aircraft_code)
        REFERENCES aircrafts(aircraft_code)
    FOREIGN KEY (arrival_airport)
        REFERENCES airports(airport_code)
```

```
    FOREIGN KEY (departure_airport)
        REFERENCES airports(airport_code)
Ссылки извне:
    TABLE "ticket_flights" FOREIGN KEY (flight_id)
        REFERENCES flights(flight_id)
```

J.4.9. Таблица `bookings.seats`

Места определяют схему салона каждой модели. Каждое место определяется своим номером (`seat_no`) и имеет закреплённый за ним класс обслуживания (`fare_conditions`) — `Economy`, `Comfort` или `Business`.

```
     Столбец     |     Тип     | Модификаторы |      Описание
-----------------+-------------+--------------+--------------------
 aircraft_code   | char(3)     | not null     | Код самолета, IATA
 seat_no         | varchar(4)  | not null     | Номер места
 fare_conditions | varchar(10) | not null     | Класс обслуживания
Индексы:
    PRIMARY KEY, btree (aircraft_code, seat_no)
Ограничения-проверки:
    CHECK (fare_conditions IN ('Economy', 'Comfort', 'Business'))
Ограничения внешнего ключа:
    FOREIGN KEY (aircraft_code)
        REFERENCES aircrafts(aircraft_code) ON DELETE CASCADE
```

J.4.10. Таблица `bookings.ticket_flights`

Перелёт соединяет билет с рейсом и идентифицируется их номерами.

Для каждого перелета указываются его стоимость (`amount`) и класс обслуживания (`fare_conditions`).

```
     Столбец     |     Тип       | Модификаторы |    Описание
-----------------+---------------+--------------+---------------------
 ticket_no       | char(13)      | not null     | Номер билета
 flight_id       | integer       | not null     | Идентификатор рейса
 fare_conditions | varchar(10)   | not null     | Класс обслуживания
 amount          | numeric(10,2) | not null     | Стоимость перелета
Индексы:
    PRIMARY KEY, btree (ticket_no, flight_id)
Ограничения-проверки:
    CHECK (amount >= 0)
    CHECK (fare_conditions IN ('Economy', 'Comfort', 'Business'))
Ограничения внешнего ключа:
    FOREIGN KEY (flight_id) REFERENCES flights(flight_id)
    FOREIGN KEY (ticket_no) REFERENCES tickets(ticket_no)
Ссылки извне:
    TABLE "boarding_passes" FOREIGN KEY (ticket_no, flight_id)
        REFERENCES ticket_flights(ticket_no, flight_id)
```

J.4.11. Таблица `bookings.tickets`

Билет имеет уникальный номер (`ticket_no`), состоящий из 13 цифр.

Билет содержит идентификатор пассажира (`passenger_id`) — номер документа, удостоверяющего личность, — его фамилию и имя (`passenger_name`) и контактную информацию (`contact_data`).

Ни идентификатор пассажира, ни имя не являются постоянными (можно поменять паспорт, можно сменить фамилию), поэтому однозначно найти все билеты одного и того же пассажира невозможно.

```
     Столбец    |     Тип     | Модификаторы |          Описание
----------------+-------------+--------------+----------------------------
 ticket_no      | char(13)    | not null     | Номер билета
 book_ref       | char(6)     | not null     | Номер бронирования
 passenger_id   | varchar(20) | not null     | Идентификатор пассажира
 passenger_name | text        | not null     | Имя пассажира
 contact_data   | jsonb       |              | Контактные данные пассажира
Индексы:
    PRIMARY KEY, btree (ticket_no)
Ограничения внешнего ключа:
    FOREIGN KEY (book_ref) REFERENCES bookings(book_ref)
Ссылки извне:
    TABLE "ticket_flights" FOREIGN KEY (ticket_no) REFERENCES tickets(ticket_no)
```

J.4.12. Представление `bookings.flights_v`

Над таблицей `flights` создано представление `flights_v`, содержащее дополнительную информацию:

- расшифровку данных об аэропорте вылета — `departure_airport`, `departure_airport_name`, `departure_city`
- расшифровку данных об аэропорте прибытия — `arrival_airport`, `arrival_airport_name`, `arrival_city`
- местное время вылета — `scheduled_departure_local`, `actual_departure_local`
- местное время прибытия — `scheduled_arrival_local`, `actual_arrival_local`
- продолжительность полета — `scheduled_duration`, `actual_duration`.

```
          Столбец           |     Тип     |              Описание
----------------------------+-------------+-------------------------------------
 flight_id                  | integer     | Идентификатор рейса
 flight_no                  | char(6)     | Номер рейса
 scheduled_departure        | timestamptz | Время вылета по расписанию
 scheduled_departure_local  | timestamp   | Время вылета по расписанию,
                            |             | местное время в пункте отправления
 scheduled_arrival          | timestamptz | Время прилёта по расписанию
 scheduled_arrival_local    | timestamp   | Время прилёта по расписанию,
                            |             | местное время в пункте прибытия
 scheduled_duration         | interval    | Планируемая продолжительность полета
 departure_airport          | char(3)     | Код аэропорта отправления
 departure_airport_name     | text        | Название аэропорта отправления
 departure_city             | text        | Город отправления
 arrival_airport            | char(3)     | Код аэропорта прибытия
 arrival_airport_name       | text        | Название аэропорта прибытия
 arrival_city               | text        | Город прибытия
 status                     | varchar(20) | Статус рейса
 aircraft_code              | char(3)     | Код самолета, IATA
 actual_departure           | timestamptz | Фактическое время вылета
 actual_departure_local     | timestamp   | Фактическое время вылета,
                            |             | местное время в пункте отправления
 actual_arrival             | timestamptz | Фактическое время прилёта
 actual_arrival_local       | timestamp   | Фактическое время прилёта,
                            |             | местное время в пункте прибытия
```

```
 actual_duration           | interval    | Фактическая продолжительность полета
```

J.4.13. Представление bookings.routes

Таблица рейсов (bookings.flights) содержит избыточность: из неё можно было бы выделить информацию о маршруте (номер рейса, аэропорты отправления и назначения), которая не зависит от конкретных дат рейсов.

Именно такая информация составляет представление routes.

```
        Столбец          |    Тип    |               Описание
-------------------------+-----------+--------------------------------------
 flight_no               | char(6)   | Номер рейса
 departure_airport       | char(3)   | Код аэропорта отправления
 departure_airport_name  | text      | Название аэропорта отправления
 departure_city          | text      | Город отправления
 arrival_airport         | char(3)   | Код аэропорта прибытия
 arrival_airport_name    | text      | Название аэропорта прибытия
 arrival_city            | text      | Город прибытия
 aircraft_code           | char(3)   | Код самолета, IATA
 duration                | interval  | Продолжительность полета
 days_of_week            | integer[] | Дни недели, когда выполняются рейсы
```

J.4.14. Функция bookings.now

Демонстрационная база содержит временной «срез» данных — так, как будто в некоторый момент была сделана резервная копия реальной системы. Например, если некоторый рейс имеет статус Departed, это означает, что в момент резервного копирования самолет вылетел и находился в воздухе.

Позиция «среза» сохранена в функции bookings.now() function. Ей можно пользоваться в запросах там, где в обычной жизни использовалась бы функция now().

Кроме того, значение этой функции определяет версию демонстрационной базы данных. Актуальная версия на текущий момент — от 15 августа 2017 г.

J.4.15. Функция bookings.lang

Некоторые поля в демонстрационной базе содержат текст на английском и русском языках. Переводы на другие языки отсутствуют, но их несложно добавить. Функция bookings.lang возвращает значение параметра *bookings.lang*, то есть язык, на котором будут выдаваться значения этих полей.

Эта функция используется в представлениях aircrafts и airports и не предназначена для непосредственного использования в запросах.

J.5. Использование

J.5.1. Схема bookings

Схема bookings содержит все объекты демонстрационной базы. При подключении к базе параметр конфигурации search_path автоматически принимает значение bookings, public, так что указывать имя схемы явно необязательно.

Однако для функции bookings.now в любом случае необходимо явно указывать схему, чтобы отличать её от стандартной функции now.

J.5.2. Переводы

По умолчанию значения различных переводимых полей выдаются на русском языке. Это поля `airport_name` и `city` представления `airports`, а также поле `model` представления `aircrafts`.

Вы можете выбрать другой язык для отображения этих полей (хотя в данной базе представлен только английский перевод). Например, для переключения на английский установите для параметра *`bookings.lang`* значение `en`. Также может быть удобно выбрать язык на уровне базы данных:

```
ALTER DATABASE demo SET bookings.lang = en;
```

Чтобы это изменение вступило в силу, необходимо переподключиться к базе данных. Другие варианты установления параметров конфигурации описаны в Разделе 18.1.

В следующих примерах для переводимых полей выбран русский язык.

J.5.3. Примеры запросов

Чтобы лучше познакомиться с содержимым демонстрационной базы данных, посмотрим на результаты нескольких простых запросов.

Результаты, представленные ниже, были получены для версии с небольшой базой данных (demo-small) от 15 августа 2017 г. Если в вашей системе запросы выдают другие данные, проверьте версию демонстрационной базы (функция `bookings.now`). Незначительные отклонения могут быть связаны с местным временем, отличным от московского, и настройками локализации.

Все рейсы выполняются несколькими типами самолетов:

```
SELECT * FROM aircrafts;
```

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 773           | Boeing 777-300       | 11100
 763           | Boeing 767-300       |  7900
 SU9           | Sukhoi SuperJet-100  |  3000
 320           | Airbus A320-200      |  5700
 321           | Airbus A321-200      |  5600
 319           | Airbus A319-100      |  6700
 733           | Boeing 737-300       |  4200
 CN1           | Cessna 208 Caravan   |  1200
 CR2           | Bombardier CRJ-200   |  2700
(9 строк)
```

Для каждого типа самолета поддерживается список мест в салоне. Например, вот где можно разместиться в небольшом самолете Cessna 208 Caravan:

```
SELECT   a.aircraft_code,
         a.model,
         s.seat_no,
         s.fare_conditions
FROM     aircrafts a
         JOIN seats s ON a.aircraft_code = s.aircraft_code
WHERE    a.model = 'Cessna 208 Caravan'
ORDER BY s.seat_no;
```

```
 aircraft_code |       model        | seat_no | fare_conditions
```

```
---------------+--------------------+---------+-----------------
 CN1           | Cessna 208 Caravan | 1A      | Economy
 CN1           | Cessna 208 Caravan | 1B      | Economy
 CN1           | Cessna 208 Caravan | 2A      | Economy
 CN1           | Cessna 208 Caravan | 2B      | Economy
 CN1           | Cessna 208 Caravan | 3A      | Economy
 CN1           | Cessna 208 Caravan | 3B      | Economy
 CN1           | Cessna 208 Caravan | 4A      | Economy
 CN1           | Cessna 208 Caravan | 4B      | Economy
 CN1           | Cessna 208 Caravan | 5A      | Economy
 CN1           | Cessna 208 Caravan | 5B      | Economy
 CN1           | Cessna 208 Caravan | 6A      | Economy
 CN1           | Cessna 208 Caravan | 6B      | Economy
(12 строк)
```

Самолеты большего размера имеют больше посадочных мест с разными классами обслуживания:

```
SELECT   s2.aircraft_code,
         string_agg (s2.fare_conditions || '(' || s2.num::text || ')',
                     ', ') as fare_conditions
FROM     (
          SELECT   s.aircraft_code, s.fare_conditions, count(*) as num
          FROM     seats s
          GROUP BY s.aircraft_code, s.fare_conditions
          ORDER BY s.aircraft_code, s.fare_conditions
         ) s2
GROUP BY s2.aircraft_code
ORDER BY s2.aircraft_code;
```

```
 aircraft_code |             fare_conditions
---------------+-----------------------------------------
 319           | Business(20), Economy(96)
 320           | Business(20), Economy(120)
 321           | Business(28), Economy(142)
 733           | Business(12), Economy(118)
 763           | Business(30), Economy(192)
 773           | Business(30), Comfort(48), Economy(324)
 CN1           | Economy(12)
 CR2           | Economy(50)
 SU9           | Business(12), Economy(85)
(9 строк)
```

База данных содержит список аэропортов практически всех крупных городов России. В большинстве городов есть только один аэропорт. Исключение составляют:

```
SELECT   a.airport_code as code,
         a.airport_name,
         a.city,
         a.coordinates
FROM     airports a
WHERE    a.city IN (
            SELECT   aa.city
            FROM     airports aa
            GROUP BY aa.city
            HAVING   COUNT(*) > 1
         )
ORDER BY a.city, a.airport_code;
```

```
 code |      airport_name      |   city    |           coordinates
------+------------------------+-----------+-------------------------
 DME  | Домодедово             | Москва    | (37.9062995910645,55.4087982177734)
 SVO  | Шереметьево            | Москва    | (37.4146,55.972599)
 VKO  | Внуково                | Москва    | (37.2615013123,55.5914993286)
 ULV  | Баратаевка             | Ульяновск | (48.2266998291,54.2682991028)
 ULY  | Ульяновск-Восточный    | Ульяновск | (48.8027000427246,54.4010009765625)
(5 строк)
```

Чтобы понять, откуда и куда можно улететь, удобно использовать материализованное представление `routes`, в котором агрегируется информация о всех рейсах. Вот, например, куда, в какие дни недели и за какое время можно долететь из Волгограда:

```
SELECT r.arrival_city as city,
       r.arrival_airport as code,
       r.arrival_airport_name as airport_name,
       r.days_of_week,
       r.duration
FROM   routes r
WHERE  r.departure_city = 'Волгоград';
```

```
    city        | code |            airport_name            |  days_of_week   |
 duration
----------------+------+------------------------------------+-----------------
+----------
 Москва         | SVO  | Шереметьево                        | {1,2,3,4,5,6,7} |
 01:15:00
 Челябинск      | CEK  | Челябинск                          | {1,2,3,4,5,6,7} |
 01:50:00
 Ростов-на-Дону | ROV  | Ростов-на-Дону                     | {1,2,3,4,5,6,7} |
 00:30:00
 Москва         | VKO  | Внуково                            | {1,2,3,4,5,6,7} |
 01:10:00
 Чебоксары      | CSY  | Чебоксары                          | {1,2,3,4,5,6,7} |
 02:45:00
 Томск          | TOF  | Богашёво                           | {1}             |
 03:50:00
(6 строк)
```

База данных была сформирована на момент времени, возвращаемый функцией `bookings.now()`:

```
SELECT bookings.now() as now;
```

```
          now
------------------------
 2017-08-15 18:00:00+03
```

Относительно именно этого момента времени все рейсы делятся на прошедшие и будущие:

```
SELECT   status,
         count(*) as count,
         min(scheduled_departure) as min_scheduled_departure,
         max(scheduled_departure) as max_scheduled_departure
FROM     flights
GROUP BY status
```

```
ORDER BY min_scheduled_departure;
```

```
  status   | count | min_scheduled_departure | max_scheduled_departure
-----------+-------+-------------------------+-------------------------
 Arrived   | 16707 | 2017-07-16 01:50:00+03  | 2017-08-15 17:25:00+03
 Cancelled |   414 | 2017-07-19 11:35:00+03  | 2017-09-14 20:55:00+03
 Departed  |    58 | 2017-08-15 09:55:00+03  | 2017-08-15 17:50:00+03
 Delayed   |    41 | 2017-08-15 15:15:00+03  | 2017-08-16 17:25:00+03
 On Time   |   518 | 2017-08-15 17:55:00+03  | 2017-08-16 18:00:00+03
 Scheduled | 15383 | 2017-08-16 18:05:00+03  | 2017-09-14 20:40:00+03
(6 строк)
```

Найдем ближайший рейс, вылетающий из Екатеринбурга в Москву. Использовать для такого запроса таблицу `flight` не очень удобно, так как в ней нет информации о городах отправления и прибытия. Поэтому воспользуемся представлением `flights_v`:

```
\x
SELECT   f.*
FROM     flights_v f
WHERE    f.departure_city = 'Екатеринбург'
AND      f.arrival_city = 'Москва'
AND      f.scheduled_departure > bookings.now()
ORDER BY f.scheduled_departure
LIMIT    1;
```

```
-[ RECORD 1 ]-------------+-----------------------------------
flight_id                | 10927
flight_no                | PG0226
scheduled_departure      | 2017-08-16 08:10:00+03
scheduled_departure_local | 2017-08-16 10:10:00
scheduled_arrival        | 2017-08-16 09:55:00+03
scheduled_arrival_local  | 2017-08-16 09:55:00
scheduled_duration       | 01:45:00
departure_airport        | SVX
departure_airport_name   | Кольцово
departure_city           | Екатеринбург
arrival_airport          | SVO
arrival_airport_name     | Шереметьево
arrival_city             | Москва
status                   | On Time
aircraft_code            | 773
actual_departure         |
actual_departure_local   |
actual_arrival           |
actual_arrival_local     |
actual_duration          |
```

Обратите внимание, что в представлении `flights_v` указано не только московское время, но и местное время в аэропортах вылета и прилета.

J.5.4. Бронирования

Каждое бронирование может включать несколько билетов, по одному на каждого пассажира. Билет, в свою очередь, может включать несколько перелетов. Полная информация о бронировании находится в трёх таблицах: `bookings`, `tickets` и `ticket_flights`.

Найдём несколько бронирований с самой высокой стоимостью:

```
SELECT   *
FROM     bookings
ORDER BY total_amount desc
LIMIT    10;
```

```
 book_ref |       book_date        | total_amount
----------+------------------------+--------------
 3B54BB   | 2017-07-05 17:08:00+03 |   1204500.00
 3AC131   | 2017-07-31 01:06:00+03 |   1087100.00
 65A6EA   | 2017-07-03 06:28:00+03 |   1065600.00
 D7E9AA   | 2017-08-08 05:29:00+03 |   1062800.00
 EF479E   | 2017-08-02 15:58:00+03 |   1035100.00
 521C53   | 2017-07-08 09:25:00+03 |    985500.00
 514CA6   | 2017-07-27 05:07:00+03 |    955000.00
 D70BD9   | 2017-07-05 12:47:00+03 |    947500.00
 EC7EDA   | 2017-07-02 16:13:00+03 |    946800.00
 8E4370   | 2017-07-28 02:04:00+03 |    945700.00
(10 строк)
```

Посмотрим, из каких билетов состоит бронирование с кодом `521C53`:

```
SELECT ticket_no,
       passenger_id,
       passenger_name
FROM   tickets
WHERE  book_ref = '521C53';
```

```
   ticket_no   | passenger_id |   passenger_name
---------------+--------------+--------------------
 0005432661914 | 8234 547529  | IVAN IVANOV
 0005432661915 | 2034 201228  | ANTONINA KUZNECOVA
(2 строки)
```

Если нас интересует, какие перелеты включены в билет Антонины Кузнецовой, то это можно узнать запросом:

```
SELECT   to_char(f.scheduled_departure, 'DD.MM.YYYY') AS when,
         f.departure_city || ' (' || f.departure_airport || ')' AS departure,
         f.arrival_city || ' (' || f.arrival_airport || ')' AS arrival,
         tf.fare_conditions AS class,
         tf.amount
FROM     ticket_flights tf
         JOIN flights_v f ON tf.flight_id = f.flight_id
WHERE    tf.ticket_no = '0005432661915'
ORDER BY f.scheduled_departure;
```

```
    when    |      departure       |       arrival        |  class   |  amount
------------+----------------------+----------------------+----------+-----------
 29.07.2017 | Москва (SVO)         | Анадырь (DYR)        | Business | 185300.00
 02.08.2017 | Анадырь (DYR)        | Khabarovsk (KHV)     | Business |  92200.00
 03.08.2017 | Хабаровск (KHV)      | Благовещенск (BQS)   | Business |  18000.00
 08.08.2017 | Благовещенск (BQS)   | Хабаровск (KHV)      | Business |  18000.00
 12.08.2017 | Хабаровск (KHV)      | Анадырь (DYR)        | Economy  |  30700.00
 17.08.2017 | Анадырь (DYR)        | Москва (SVO)         | Business | 185300.00
(6 строк)
```

Как видим, высокая стоимость бронирования объясняется большим количеством перелётов на дальние расстояния бизнес-классом.

Часть перелётов в этом билете имеет более ранние даты, чем значение `bookings.now()`: это значит, что они уже выполнены. А последний полет ещё предстоит. После регистрации на рейс выписывается посадочный талон с указанием места в самолете. Мы можем посмотреть какие именно места занимала Антонина (обратите внимание на внешнее левое соединение с таблицей `boarding_passes`):

```
SELECT   to_char(f.scheduled_departure, 'DD.MM.YYYY') AS when,
         f.departure_city || ' (' || f.departure_airport || ')' AS departure,
         f.arrival_city || ' (' || f.arrival_airport || ')' AS arrival,
         f.status,
         bp.seat_no
FROM     ticket_flights tf
         JOIN flights_v f ON tf.flight_id = f.flight_id
         LEFT JOIN boarding_passes bp ON tf.flight_id = bp.flight_id
                                     AND tf.ticket_no = bp.ticket_no
WHERE    tf.ticket_no = '0005432661915'
ORDER BY f.scheduled_departure;
```

```
    when    |      departure       |       arrival        |  status   | seat_no
------------+----------------------+----------------------+-----------+---------
 29.07.2017 | Москва (SVO)         | Анадырь (DYR)        | Arrived   | 5C
 02.08.2017 | Анадырь (DYR)        | Хабаровск (KHV)      | Arrived   | 1D
 03.08.2017 | Хабаровск (KHV)      | Благовещенск (BQS)   | Arrived   | 2C
 08.08.2017 | Благовещенск(BQS)    | Хабаровск(KHV)       | Arrived   | 2D
 12.08.2017 | Хабаровск(KHV)       | Анадырь(DYR)         | Arrived   | 20B
 17.08.2017 | Анадырь(DYR)         | Москва(SVO)          | Scheduled |
(6 строк)
```

J.5.5. Новое бронирование

Попробуем отправить Александра Николаевича Радищева по маршруту, который принес ему известность. Разумеется бесплатно и бизнес-классом. Предварительно найдем «завтрашний» рейс, а также позаботимся о возвращении через неделю.

```
BEGIN;

INSERT INTO bookings (book_ref, book_date, total_amount)
VALUES      ('_QWE12', bookings.now(), 0);

INSERT INTO tickets (ticket_no, book_ref, passenger_id, passenger_name)
VALUES      ('_000000000001', '_QWE12', '1749 051790', 'ALEKSANDR RADISHCHEV');

INSERT INTO ticket_flights (ticket_no, flight_id, fare_conditions, amount)
VALUES      ('_000000000001', 8525, 'Business', 0),
            ('_000000000001', 4967, 'Business', 0);

COMMIT;
```

Мы начинаем идентификаторы с подчёркивания, чтобы не пересекаться с диапазоном значений, присутствующих в базе.

Сразу зарегистрируемся на завтрашний рейс:

```
INSERT INTO boarding_passes (ticket_no, flight_id, boarding_no, seat_no)
VALUES      ('_000000000001', 8525, 1, '1A');
```

Проверим информацию о созданном бронировании:

```
SELECT   b.book_ref,
         t.ticket_no,
         t.passenger_id,
         t.passenger_name,
         tf.fare_conditions,
         tf.amount,
         f.scheduled_departure_local,
         f.scheduled_arrival_local,
         f.departure_city || ' (' || f.departure_airport || ')' AS departure,
         f.arrival_city || ' (' || f.arrival_airport || ')' AS arrival,
         f.status,
         bp.seat_no
FROM     bookings b
         JOIN tickets t ON b.book_ref = t.book_ref
         JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
         JOIN flights_v f ON tf.flight_id = f.flight_id
         LEFT JOIN boarding_passes bp ON tf.flight_id = bp.flight_id
                                     AND tf.ticket_no = bp.ticket_no
WHERE    b.book_ref = '_QWE12'
ORDER BY t.ticket_no, f.scheduled_departure;
```

```
-[ RECORD 1 ]-------------+--------------------
book_ref                  | _QWE12
ticket_no                 | _000000000001
passenger_id              | 1749 051790
passenger_name            | ALEKSANDR RADISHCHEV
fare_conditions           | Business
amount                    | 0.00
scheduled_departure_local | 2017-08-16 09:45:00
scheduled_arrival_local   | 2017-08-16 10:35:00
departure                 | Санкт-Петербург (LED)
arrival                   | Москва (SVO)
status                    | On Time
seat_no                   | 1A
-[ RECORD 2 ]-------------+--------------------
book_ref                  | _QWE12
ticket_no                 | _000000000001
passenger_id              | 1749 051790
passenger_name            | ALEKSANDR RADISHCHEV
fare_conditions           | Business
amount                    | 0.00
scheduled_departure_local | 2017-08-23 10:20:00
scheduled_arrival_local   | 2017-08-23 11:10:00
departure                 | Москва (SVO)
arrival                   | Санкт-Петербург (LED)
status                    | Scheduled
seat_no                   |
```

Надеемся, что эти несколько простых примеров помогли составить представление о содержимом демонстрационной базы данных.

Приложение K. Сокращения

Ниже перечислены сокращения, часто используемые в документации Postgres Pro, и в обсуждениях, связанных с Postgres Pro.

ANSI

American National Standards Institute, Американский национальный институт стандартов

API

Application Programming Interface, *Интерфейс программирования приложений*

ASCII

American Standard Code for Information Interchange, *Американский стандартный код для обмена информацией*

BKI

Backend Interface, Внутренний интерфейс

CA

Certificate Authority, *Центр сертификации*

CIDR

Classless Inter-Domain Routing, *Бесклассовая междоменная маршрутизация*

CPAN

Comprehensive Perl Archive Network, Всеобъемлющая сеть архивов Perl

CRL

Certificate Revocation List, Список отозванных сертификатов

CSV

Comma Separated Values, *Значения, разделённые запятыми*

CTE

Common Table Expression, Общее табличное выражение

CVE

Common Vulnerabilities and Exposures, Общие уязвимости и риски

DBA

Database Administrator, *Администратор баз данных*

DBI

Database Interface (Perl), Интерфейс баз данных (Perl)

DBMS

Database Management System, *Система управления базами данных*

DDL

Data Definition Language, *Язык описания данных*, включающий такие команды SQL, как `CREATE TABLE`, `ALTER USER`

DML

Data Manipulation Language, *Язык обработки данных*, включающий такие SQL-команды, как `INSERT`, `UPDATE`, `DELETE`

DST

Daylight Saving Time, *Летнее время*

ECPG

Embedded C for Postgres Pro, Встроенный C для Postgres Pro

ESQL

Embedded SQL, *Встраиваемый SQL*

FAQ

Frequently Asked Questions, *Часто задаваемые вопросы*

FSM

Free Space Map, Карта свободного пространства

GEQO

Genetic Query Optimizer, Генетический оптимизатор запросов

GIN

Generalized Inverted Index, Обобщённый инвертированный индекс

GiST

Generalized Search Tree, Обобщённое дерево поиска

Git

Git

GMT

Greenwich Mean Time, *Среднее время по Гринвичу*

GSSAPI

Generic Security Services Application Programming Interface, *Универсальный интерфейс программирования приложений служб безопасности*

GUC

Grand Unified Configuration, Главная унифицированная конфигурация, подсистема Postgres Pro, управляющая конфигурацией сервера

HBA

Host-Based Authentication, Аутентификация по сетевым узлам

HOT

Heap-Only Tuples, «Кортежи только в куче»

IEC

International Electrotechnical Commission, *Международная электротехническая комиссия*

IEEE

Institute of Electrical and Electronics Engineers, Институт инженеров по электротехнике и электронике

IPC

Inter-Process Communication, *Межпроцессное взаимодействие*

ISO

International Organization for Standardization, Международная организация по стандартизации

ISSN

International Standard Serial Number, *Международный стандартный серийный номер*

JDBC

Java Database Connectivity, *Соединение с базами данных на Java*

LDAP

Lightweight Directory Access Protocol, *Облегчённый протокол доступа к каталогам*

LSN

Log Sequence Number, Последовательный номер в журнале; см. `pg_lsn` и Внутреннее устройство WAL.

MSVC

Microsoft Visual C

MVCC

Multi-Version Concurrency Control, Многоверсионное управление конкурентным доступом

NLS

National Language Support, *Поддержка национальных языков*

ODBC

Open Database Connectivity, *Открытое соединение с базами данных*

OID

Object Identifier, Идентификатор объекта

OLAP

Online Analytical Processing, *Непосредственная аналитическая обработка*

OLTP

Online Transaction Processing, *Непосредственная транзакционная обработка*

ORDBMS

Object-Relational Database Management System, *Объектно-реляционная система управления базами данных*

PAM

Pluggable Authentication Modules, *Подключаемые модули аутентификации*

PGSQL

Postgres Pro

PGXS

Postgres Pro Extension System, Система расширений Postgres Pro

PID

Process Identifier, *Идентификатор процесса*

PITR

Point-In-Time Recovery, Восстановление на момент времени (Непрерывное архивирование)

PL

Procedural Languages, Процедурные языки (на стороне сервера)

POSIX

Portable Operating System Interface, *Переносимый интерфейс операционных систем*

RDBMS

Relational Database Management System, *Реляционная система управления базами данных*

RFC

Request For Comments, *Рабочее предложение*

SGML

Standard Generalized Markup Language, *Стандартный обобщённый язык разметки*

SPI

Server Programming Interface, Интерфейс программирования сервера

SP-GiST

Space-Partitioned Generalized Search Tree, Обобщённое дерево поиска с разбиением пространства

SQL

Structured Query Language, *Язык структурированных запросов*

SRF

Set-Returning Function, Функция, возвращающая множество

SSH

Secure Shell, *Защищённая оболочка*

SSL

Secure Sockets Layer, *Уровень защищённых сокетов*

SSPI

Security Support Provider Interface, Интерфейс поставщика поддержки безопасности

SYSV

Unix System V

TCP/IP

Transmission Control Protocol (TCP) / Internet Protocol (IP), *Протокол управления передачей/ Межсетевой протокол*

TID

Tuple Identifier, Идентификатор кортежа

TLS

Transport Layer Security, *Протокол защиты транспортного уровня*

TOAST

The Oversized-Attribute Storage Technique, Методика хранения сверхбольших атрибутов

TPC

Transaction Processing Performance Council, Совет по оценке производительности обработки транзакций

URL

Uniform Resource Locator, *Универсальный указатель ресурса*

UTC

Coordinated Universal Time, *Универсальное координированное время*

UTF

Unicode Transformation Format, Формат преобразования Unicode

UTF8

Eight-Bit Unicode Transformation Format, *Восьмибитный формат преобразования Unicode*

UUID

Universally Unique Identifier, Универсальный уникальный идентификатор

WAL

Write-Ahead Log, Журнал предзаписи

XID

Transaction Identifier, Идентификатор транзакции

XML

Extensible Markup Language, *Расширяемый язык разметки*

Приложение L. Устаревшая или переименованная функциональность

Иногда в PostgreSQL меняется функциональность, переименовываются функции, параметры и файлы, или документация перемещается в другое место. Данный раздел помогает читателям документации старых версий, а также пришедшим по внешним ссылкам, найти необходимую им информацию в новом месте.

L.1. `pg_xlogdump` переименована в `pg_waldump`

В PostgreSQL 9.6 и более ранних версиях существовала команда `pg_xlogdump` (предназначенная для чтения файлов журнала предзаписи (WAL)). Эта команда была переименована в `pg_waldump`, о ней рассказывается в описании pg_waldump, а информация о данном изменении изложена в Замечаниях к выпуску PostgreSQL 10.

L.2. `pg_resetxlog` переименована в `pg_resetwal`

В PostgreSQL 9.6 и более ранних версиях существовала команда `pg_resetxlog` (предназначенная для восстановления файлов журнала предзаписи (WAL)). Эта команда была переименована в `pg_resetwal`; о ней рассказывается в описании pg_resetwal, а информация о данном изменении представлена в Замечаниях к выпуску PostgreSQL 10.

L.3. `pg_receivexlog` переименована в `pg_receivewal`

В PostgreSQL 9.6 и более ранних версиях существовала команда `pg_receivexlog` (предназначенная для получения файлов журнала предзаписи (WAL)). Эта команда была переименована в `pg_receivewal`; о ней рассказывается в описании pg_receivewal, а информация о данном изменении представлена в Замечаниях к выпуску PostgreSQL 10.

Библиография

Здесь представлены ссылки на справочные материалы и книги, посвящённые SQL и PostgreSQL.

Некоторые статьи и технические описания, написанные первой командой разработки POSTGRES, можно найти на *сайте* факультета компьютерных наук Калифорнийского университета в Беркли.

Справочники по языку SQL

[bowman01] *The Practical SQL Handbook*. Using SQL Variants. Fourth Edition. Judith Bowman, Sandra Emerson, Marcy Darnovsky. ISBN 0-201-70309-2. Addison-Wesley Professional. 2001.

[date97] *A Guide to the SQL Standard*. A user's guide to the standard database language SQL. Fourth Edition. C. J. Date Hugh Darwen. ISBN 0-201-96426-0. Addison-Wesley. 1997.

[date04] *An Introduction to Database Systems*. Eighth Edition. C. J. Date. ISBN 0-321-19784-4. Addison-Wesley. 2003.

[elma04] *Fundamentals of Database Systems*. Fourth Edition. Ramez Elmasri Shamkant Navathe. ISBN 0-321-12226-7. Addison-Wesley. 2003.

[melt93] *Understanding the New SQL*. A complete guide. Jim Melton Alan R. Simon. ISBN 1-55860-245-3. Morgan Kaufmann. 1993.

[ull88] *Principles of Database and Knowledge-Base Systems*. Classical Database Systems. Jeffrey D. Ullman. Volume 1. Computer Science Press. 1988.

Документация, посвящённая PostgreSQL

[sim98] *Enhancement of the ANSI SQL Implementation of PostgreSQL*. Stefan Simkovics. Department of Information Systems, Vienna University of Technology. Vienna, Austria. November 29, 1998.

[yu95] *The Postgres95. User Manual*. A. Yu J. Chen. University of California. Berkeley, California. Sept. 5, 1995.

[fong] *The design and implementation of the POSTGRES query optimizer*. Zelaine Fong. University of California, Berkeley, Computer Science Department.

Заметки и статьи

[ports12] «*Serializable Snapshot Isolation in PostgreSQL*». D. Ports K. Grittner. VLDB Conference, August 2012.

[berenson95] «*A Critique of ANSI SQL Isolation Levels*». H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil. ACM-SIGMOD Conference on Management of Data, June 1995.

[olson93] *Partial indexing in POSTGRES: research project*. Nels Olson. UCB Engin T7.49.1993 O676. University of California. Berkeley, California. 1993.

[ong90] «A Unified Framework for Version Modeling Using Production Rules in a Database System». L. Ong J. Goh. *ERL Technical Memorandum M90/33*. University of California. Berkeley, California. April, 1990.

[rowe87] «*The POSTGRES data model*». L. Rowe M. Stonebraker. VLDB Conference, Sept. 1987.

[seshadri95] «*Generalized Partial Indexes*». P. Seshadri A. Swami. Eleventh International Conference on Data Engineering, 6-10 March 1995. Cat. No.95CH35724. IEEE Computer Society Press. Los Alamitos, California. 1995. 420-7.

[ston86] «*The design of POSTGRES*». M. Stonebraker L. Rowe. ACM-SIGMOD Conference on Management of Data, May 1986.

[ston87a] «The design of the POSTGRES. rules system». M. Stonebraker, E. Hanson, C. H. Hong. IEEE Conference on Data Engineering, Feb. 1987.

[ston87b] «*The design of the POSTGRES storage system*». M. Stonebraker. VLDB Conference, Sept. 1987.

[ston89] «*A commentary on the POSTGRES rules system*». M. Stonebraker, M. Hearst, S. Potamianos. *SIGMOD Record 18(3)*. Sept. 1989.

[ston89b] «*The case for partial indexes*». M. Stonebraker. *SIGMOD Record 18(4)*. Dec. 1989. 4-11.

[ston90a] «*The implementation of POSTGRES*». M. Stonebraker, L. A. Rowe, M. Hirohama. *Transactions on Knowledge and Data Engineering 2(1)*. IEEE. March 1990.

[ston90b] «*On Rules, Procedures, Caching and Views in Database Systems*». M. Stonebraker, A. Jhingran, J. Goh, S. Potamianos. ACM-SIGMOD Conference on Management of Data, June 1990.

Предметный указатель

Символы

$, 33
$libdir, 1003
$libdir/plugins, 551, 1627
*, 107
.pgpass, 798
.pg_service.conf, 799
::, 39
_PG_fini, 1003
_PG_init, 1003
_PG_output_plugin_init, 1274
«грязное» чтение, 414
Булевы
  операторы (см. операторы, логические)
Ведение журнала
  файл current_logfiles и функция pg_current_logfile, 311
  функция pg_current_logfile, 311
Григорианский календарь, 2165
Источники репликации, 1279
Карта видимости, 2134
Карта свободного пространства, 2133
Каскадная репликация, 640
Логическое декодирование, 1270, 1272
Многоверсионное управление конкурентным доступом, 414
Отслеживание прогресса репликации, 1279
Оценка количества строк
  планировщик, 2141
Потоковая репликация, 640
Сериализуемая изоляция снимков, 414
Сетевые файловые системы, 470
Сетевые хранилища (NAS) (см. Сетевые файловые системы)
Символы Unicode
  в идентификаторах, 24
Синхронная репликация, 640
Словари Hunspell, 2452
Слой инициализации, 2134
Слоты репликации
  Потоковая репликация, 648
Спецкоды Unicode
  в строковых константах, 26
Триггер
  Аргументы для триггерных функций, 1066
  на C, 1067
Файл сопоставления имён пользователей, 571
Фоновые рабочие процессы, 1267
Юлианская дата, 2166
автоочистка
  общая информация, 618
  параметры конфигурации, 542
автофиксация
  массовая загрузка данных, 446
агрегатная функция, 11
  вспомогательные функции, 1029
  вызов, 35
  движущийся агрегат, 1024
  полиморфная, 1026
  пользовательская, 1023
  с переменными аргументами, 1026
  сортирующая, 1027
  частичное агрегирование, 1028
агрегатные функции
  встроенные, 290
активность базы данных
  мониторинг, 668
аномалия сериализации, 415, 418
анонимные блоки кода, 1551
асинхронная фиксация данных, 725
аутентификация BSD, 581
аутентификация клиентского приложения, 564
баз данных
  право для создания, 584
база данных, 590
  создание, 2
базовый тип, 983
балансировка нагрузки, 640
битовая строка
  тип данных, 146
битовая строковая
  константа, 28
битовые строки
  функции, 207
блокировка
  мониторинг, 706
  рекомендательная, 425
большой объект, 817
быстрый путь, 779
версия, 4, 312
  совместимость, 485
взаимоблокировка, 424
  тайм-аут, 553
включение
  в файл конфигурации, 497
владелец, 60
внешнее соединение, 94
внешний ключ, 14, 54
  ссылающийся на себя, 54
восстановление на момент времени, 623
время
  константы, 133
  текущее, 242
  формат вывода, 134
    (см. также форматирование)
встраиваемый SQL
  в C, 828
выбор поля, 33
вывод ошибок
  в PL/pgSQL, 1145

выключение, 484
выражение
  порядок вычисления, 43
  синтаксис, 32
выражение значения, 32
выходной список, 1083
вычисляемое поле, 171
вычитание множеств, 109
генетическая оптимизация запросов, 527
гипотезирующие агрегатные функции
  встроенные, 296
глобальные данные
  в PL/Python, 1203
  в PL/Tcl, 1173
горячий резерв, 640
группировка, 102
дата
  константы, 133
  текущая, 242
  формат вывода, 134
    (см. также форматирование)
двоичная строка
  длина, 207
двоичные данные, 127
  функции, 205
двоичные строки
  конкатенация, 205
дерево запроса, 1082
диапазонный тип, 172
  индексы, 176
  исключение, 177
дизъюнкция, 182
динамическая загрузка, 552, 1003
дисковое пространство, 612
дисковое устройство, 730
дисперсия, 294
  для совокупности, 294
  по выборке, 295
длина
  двоичной строки (см. двоичные строки, длина)
  строки символов (см. строка символов, длина)
добавление, 89
доверенный
  PL/Perl, 1191
документ
  поиск текста, 378
доступная роль, 947
дублирование, 9, 108
естественное соединение, 95
журнал сервера, 530
  обслуживание файла журнала, 620
журнал событий
  журнал событий, 493
журнал транзакций (см. WAL)
загрузка системы
  запуск сервера, 471
задержка, 243
заключение строк в доллары, 27
запись
  функций, 44
запрос, 8, 93
зацикливание
  идентификаторов мультитранзакций, 618
  идентификаторов транзакций, 615
защита на уровне строк, 61
значение NULL
  в libpq, 768
  с ограничениями уникальности, 53
  значение по умолчанию, 48
  с ограничениями-проверками, 50
значение по умолчанию, 48
  изменение, 59
значимые цифры, 550
идентификатор
  длина, 24
  синтаксис, 23
идентификатор объекта
  тип данных, 178
идентификатор транзакции
  зацикливание, 615
иерархическая база данных, 6
изменение, 90
изменчивость
  функций, 1000
изменяемые представления, 1539
изоляция транзакций, 414
имена часовых поясов, 550
имя
  неполное, 68
  полное, 67
  синтаксис, 23
имя компьютера, 747
индекс, 362, 2471
  B-дерево, 363
  BRIN, 364, 2121
  по выражению, 368
  GIN, 364, 2114
    текстовый поиск, 409
  GiST, 363, 2093
    текстовый поиск, 409
  SP-GiST, 364, 2104
  для пользовательского типа данных, 1039
  и ORDER BY, 366
  неблокирующее построение, 1442
  объединение нескольких индексов, 367
  по хешу, 363, 2126
  сканирование только индекса, 373
  составной, 365
  уникальный, 367
  частичный, 368
индекс элемента, 33
индексы
  блокировки, 428
  контроль использования, 375
интервал
  формат вывода, 138

(см. также форматирование)
интерфейсы
поддерживаемые отдельно, 2720
информационная схема, 924
исключение по ограничению, 84, 528
исключения
в PL/pgSQL, 1136
в PL/Tcl, 1178
использование диска, 720
история
PostgreSQL, xxiii
кавычки
и идентификаторы, 24
спецсимволы, 25
класс операторов, 371, 1040
кластер
баз данных (см. кластер баз данных)
кластер баз данных, 6, 469
кластеризация, 640
ключевое слово
синтаксис, 23
список, 2168
ковариация
выборки, 293
совокупности, 293
коды ошибок
libpq, 763
список, 2150
команда TABLE, 1661
комментарии
к объектам баз данных, 322
комментарий
в SQL, 30
компиляция
приложений с libpq, 805
компонент, 23
конвейеризация
спецификация протокола, 2003
конкатенация значений tsvector, 389
конкурентный доступ, 414
константа, 25
контекст памяти
в SPI, 1253
контрольная точка, 726
конфигурация
восстановления
резервного сервера, 664
сервера
функции, 327
конъюнкция, 182
корреляция, 293
в планировщике запросов, 442
круг, 143, 248
куда протоколировать, 530
курсор
CLOSE, 1378
DECLARE, 1543
FETCH, 1605
MOVE, 1631
в PL/pgSQL, 1140
показ плана запроса, 1600
левое соединение, 95
линейная регрессия, 293
линии времени, 623
логический
тип данных, 139
локаль, 470, 596
массив, 158
ввод/вывод, 165
изменение, 162
константа, 159
конструктор, 40
обращение, 160
объявление, 158
поиск, 164
массив элементов
пользовательского типа, 1032
материализованные представления, 1976
реализация через правила, 1090
медиана, 36
(см. также процентиль)
метка (см. псевдоним)
метод TABLESAMPLE, 2063
метод извлечения выборки таблицы, 2063
многоугольник, 143, 248
мода
статистическая функция, 295
мониторинг
активность базы данных, 668
набор символов, 550, 558, 604
наклон линии регрессии, 294
нарезанный хлеб (см. TOAST)
наследование, 19, 71
настройка
сервера, 495
не число
double precision, 122
numeric (тип данных), 121
неблокирующее соединение, 740, 773
неповторяемое чтение, 414
неполное имя, 68
непрерывное архивирование, 623
на резервном сервере, 653
область данных (см. кластер баз данных)
обновление, 485
обработка замечаний
в libpq, 790
обработчик замечаний, 790
обратное распределение, 295
обслуживание, 611
общее табличное выражение (см. WITH)
объединение множеств, 109
объектно-ориентированная база данных, 6
обёртка сторонних данных
обработчик, 2044
ограничение, 49

NOT NULL, 51
внешний ключ, 54
добавление, 59
имя, 49
исключение, 56
первичный ключ, 53
проверка, 49
удаление, 59
уникальности, 52
ограничение NOT NULL, 51
ограничение уникальности, 52
ограничение-исключение, 56
ограничение-проверка, 49
оконная функция, 17
вызов, 37
порядок выполнения, 107
оконные функции
встроенные, 298
оператор, 182
вызов, 34
логический, 182
пользовательский, 1034
приоритеты, 30
разрешение типов при вызове, 351
синтаксис, 29
оператор упорядочивания, 1050
операции над множествами, 109
отказоустойчивость, 640
откладываемая транзакция, 547
выбор по умолчанию, 546
установка, 1692
отличительный блок, 1003
отмена
SQL-команд, 778
отношение, 6
отработка отказа, 640
отрезок, 142
отрицание, 182
оценка числа строк
многовариантная, 2146
очистка, 611
параллельный запрос, 450
параметр
синтаксис, 33
параметр восстановления archive_cleanup_command, 664
параметр восстановления primary_conninfo, 666
параметр восстановления primary_slot_name, 667
параметр восстановления recovery_end_command, 665
параметр восстановления recovery_min_apply_delay, 667
параметр восстановления recovery_target, 665
параметр восстановления recovery_target_action, 666
параметр восстановления recovery_target_inclusive, 666
параметр восстановления recovery_target_lsn, 665
параметр восстановления recovery_target_name, 665
параметр восстановления recovery_target_time, 665
параметр восстановления recovery_target_timeline, 666
параметр восстановления recovery_target_xid, 665
параметр восстановления restore_command, 664
параметр восстановления standby_mode, 666
параметр восстановления trigger_file, 667
параметр конфигурации allow_in_place_tablespaces, 559
параметр конфигурации allow_system_table_mods, 559
параметр конфигурации application_name, 535
параметр конфигурации archive_command, 518
параметр конфигурации archive_mode, 518
параметр конфигурации archive_timeout, 518
параметр конфигурации array_nulls, 554
параметр конфигурации authentication_timeout, 502
параметр конфигурации auth_delay.milliseconds, 2387
параметр конфигурации autovacuum, 542
параметр конфигурации autovacuum_analyze_scale_factor, 543
параметр конфигурации autovacuum_analyze_threshold, 543
параметр конфигурации autovacuum_freeze_max_age, 543
параметр конфигурации autovacuum_max_workers, 542
параметр конфигурации autovacuum_multixact_freeze_max_age, 543
параметр конфигурации autovacuum_naptime, 542
параметр конфигурации autovacuum_vacuum_cost_delay, 544
параметр конфигурации autovacuum_vacuum_cost_limit, 544
параметр конфигурации autovacuum_vacuum_scale_factor, 543
параметр конфигурации autovacuum_vacuum_threshold, 543
параметр конфигурации autovacuum_work_mem, 507
параметр конфигурации auto_explain.log_analyze, 2388
параметр конфигурации auto_explain.log_buffers, 2388
параметр конфигурации auto_explain.log_format, 2389
параметр конфигурации auto_explain.log_min_duration, 2388

параметр конфигурации auto_explain.log_nested_statements, 2389
параметр конфигурации auto_explain.log_timing, 2388
параметр конфигурации auto_explain.log_triggers, 2388
параметр конфигурации auto_explain.log_verbose, 2388
параметр конфигурации auto_explain.sample_rate, 2389
параметр конфигурации backend_flush_after, 511
параметр конфигурации backslash_quote, 554
параметр конфигурации bgwriter_delay, 509
параметр конфигурации bgwriter_flush_after, 510
параметр конфигурации bgwriter_lru_maxpages, 509
параметр конфигурации bgwriter_lru_multiplier, 510
параметр конфигурации block_size, 557
параметр конфигурации bonjour, 501
параметр конфигурации bonjour_name, 501
параметр конфигурации bytea_output, 548
параметр конфигурации checkpoint_completion_target, 517
параметр конфигурации checkpoint_flush_after, 517
параметр конфигурации checkpoint_timeout, 517
параметр конфигурации checkpoint_warning, 517
параметр конфигурации check_function_bodies, 546
параметр конфигурации client_encoding, 550
параметр конфигурации client_min_messages, 544
параметр конфигурации cluster_name, 540
параметр конфигурации commit_delay, 516
параметр конфигурации commit_siblings, 517
параметр конфигурации config_file, 499
параметр конфигурации constraint_exclusion, 528
параметр конфигурации cpu_index_tuple_cost, 526
параметр конфигурации cpu_operator_cost, 526
параметр конфигурации cpu_tuple_cost, 526
параметр конфигурации current_logfiles и log_destination, 530
параметр конфигурации cursor_tuple_fraction, 529
параметр конфигурации data_checksums, 557
параметр конфигурации data_directory, 499
параметр конфигурации data_sync_retry, 557
параметр конфигурации DateStyle, 549
параметр конфигурации db_user_namespace, 504
параметр конфигурации deadlock_timeout, 553
параметр конфигурации debug_assertions, 557
параметр конфигурации debug_deadlocks, 561
параметр конфигурации debug_pretty_print, 535
параметр конфигурации debug_print_parse, 535
параметр конфигурации debug_print_plan, 535
параметр конфигурации debug_print_rewritten, 535
параметр конфигурации default_statistics_target, 528
параметр конфигурации default_tablespace, 545
параметр конфигурации default_text_search_config, 551
параметр конфигурации default_transaction_deferrable, 546
параметр конфигурации default_transaction_isolation, 546
параметр конфигурации default_transaction_read_only, 546
параметр конфигурации default_with_oids, 555
параметр конфигурации dynamic_library_path, 552
параметр конфигурации dynamic_shared_memory_type, 507
параметр конфигурации effective_cache_size, 527
параметр конфигурации effective_io_concurrency, 510
параметр конфигурации enable_bitmapscan, 524
параметр конфигурации enable_gathermerge, 524
параметр конфигурации enable_hashagg, 524
параметр конфигурации enable_hashjoin, 524
параметр конфигурации enable_indexonlyscan, 524
параметр конфигурации enable_indexscan, 524
параметр конфигурации enable_material, 525
параметр конфигурации enable_mergejoin, 525
параметр конфигурации enable_nestloop, 525
параметр конфигурации enable_seqscan, 525
параметр конфигурации enable_sort, 525
параметр конфигурации enable_tidscan, 525
параметр конфигурации escape_string_warning, 555
параметр конфигурации event_source, 533
параметр конфигурации exit_on_error, 557
параметр конфигурации external_pid_file, 499
параметр конфигурации extra_float_digits, 550
параметр конфигурации force_parallel_mode, 529
параметр конфигурации from_collapse_limit, 529
параметр конфигурации fsync, 513
параметр конфигурации full_page_writes, 515
параметр конфигурации geqo, 527
параметр конфигурации geqo_effort, 527
параметр конфигурации geqo_generations, 528
параметр конфигурации geqo_pool_size, 528
параметр конфигурации geqo_seed, 528
параметр конфигурации geqo_selection_bias, 528
параметр конфигурации geqo_threshold, 527
параметр конфигурации gin_fuzzy_search_limit, 553
параметр конфигурации gin_pending_list_limit, 549
параметр конфигурации hba_file, 499
параметр конфигурации hot_standby, 522

параметр конфигурации hot_standby_feedback, 523
параметр конфигурации huge_pages, 505
параметр конфигурации ident_file, 499
параметр конфигурации idle_in_transaction_session_timeout, 547
параметр конфигурации ignore_checksum_failure, 561
параметр конфигурации ignore_system_indexes, 559
параметр конфигурации integer_datetimes, 558
параметр конфигурации IntervalStyle, 549
параметр конфигурации join_collapse_limit, 529
параметр конфигурации krb_caseins_users, 504
параметр конфигурации krb_server_keyfile, 504
параметр конфигурации lc_collate, 558
параметр конфигурации lc_ctype, 558
параметр конфигурации lc_messages, 550
параметр конфигурации lc_monetary, 550
параметр конфигурации lc_numeric, 550
параметр конфигурации lc_time, 550
параметр конфигурации listen_addresses, 500
параметр конфигурации local_preload_libraries, 551
параметр конфигурации lock_timeout, 547
параметр конфигурации logging_collector, 531
параметр конфигурации log_autovacuum_min_duration, 542
параметр конфигурации log_btree_build_stats, 561
параметр конфигурации log_checkpoints, 535
параметр конфигурации log_connections, 535
параметр конфигурации log_destination, 530
параметр конфигурации log_directory, 531
параметр конфигурации log_disconnections, 536
параметр конфигурации log_duration, 536
параметр конфигурации log_error_verbosity, 536
параметр конфигурации log_executor_stats, 542
параметр конфигурации log_filename, 531
параметр конфигурации log_file_mode, 532
параметр конфигурации log_hostname, 536
параметр конфигурации log_line_prefix, 536
параметр конфигурации log_lock_waits, 538
параметр конфигурации log_min_duration_statement, 534
параметр конфигурации log_min_error_statement, 533
параметр конфигурации log_min_messages, 533
параметр конфигурации log_parser_stats, 542
параметр конфигурации log_planner_stats, 542
параметр конфигурации log_replication_commands, 539
параметр конфигурации log_rotation_age, 532
параметр конфигурации log_rotation_size, 532
параметр конфигурации log_statement, 538
параметр конфигурации log_statement_stats, 542
параметр конфигурации log_temp_files, 539
параметр конфигурации log_timezone, 539
параметр конфигурации log_truncate_on_rotation, 532
параметр конфигурации lo_compat_privileges, 555
параметр конфигурации maintenance_work_mem, 506
параметр конфигурации max_connections, 500
параметр конфигурации max_files_per_process, 508
параметр конфигурации max_function_args, 558
параметр конфигурации max_identifier_length, 558
параметр конфигурации max_index_keys, 558
параметр конфигурации max_locks_per_transaction, 553
параметр конфигурации max_logical_replication_workers, 523
параметр конфигурации max_parallel_workers, 511
параметр конфигурации max_parallel_workers_per_gather, 511
параметр конфигурации max_pred_locks_per_page, 554
параметр конфигурации max_pred_locks_per_relation, 554
параметр конфигурации max_pred_locks_per_transaction, 554
параметр конфигурации max_prepared_transactions, 506
параметр конфигурации max_replication_slots, 519
параметр конфигурации max_stack_depth, 507
параметр конфигурации max_standby_archive_delay, 522
параметр конфигурации max_standby_streaming_delay, 522
параметр конфигурации max_sync_workers_per_subscription, 524
параметр конфигурации max_wal_senders, 519
параметр конфигурации max_wal_size, 517
параметр конфигурации max_worker_processes, 511
параметр конфигурации min_parallel_index_scan_size, 527
параметр конфигурации min_parallel_table_scan_size, 527
параметр конфигурации min_wal_size, 518
параметр конфигурации nul_byte_replacement_on_import, 556
параметр конфигурации old_snapshot_threshold, 512
параметр конфигурации operator_precedence_warning, 555
параметр конфигурации parallel_setup_cost, 526
параметр конфигурации parallel_tuple_cost, 526
параметр конфигурации password_encryption, 504

параметр конфигурации pg_trgm.similarity_threshold, 2536
параметр конфигурации pg_trgm.word_similarity_threshold, 2536
параметр конфигурации plperl.on_init, 1194
параметр конфигурации plperl.on_plperlu_init, 1195
параметр конфигурации plperl.on_plperl_init, 1195
параметр конфигурации plperl.use_strict, 1195
параметр конфигурации plpgsql.check_asserts, 1147
параметр конфигурации plpgsql.variable_conflict, 1156
параметр конфигурации pltcl.start_proc, 1180
параметр конфигурации pltclu.start_proc, 1180
параметр конфигурации port, 500
параметр конфигурации post_auth_delay, 559
параметр конфигурации pre_auth_delay, 559
параметр конфигурации quote_all_identifiers, 555
параметр конфигурации random_page_cost, 526
параметр конфигурации replacement_sort_tuples, 506
параметр конфигурации restart_after_crash, 557
параметр конфигурации row_security, 545
параметр конфигурации search_path, 69, 544
  использование в защищённых функциях, 1436
параметр конфигурации segment_size, 558
параметр конфигурации sepgsql.debug_audit, 2562
параметр конфигурации sepgsql.permissive, 2561
параметр конфигурации seq_page_cost, 525
параметр конфигурации server_encoding, 558
параметр конфигурации server_version, 558
параметр конфигурации server_version_num, 558
параметр конфигурации session_preload_libraries, 551
параметр конфигурации session_replication_role, 547
параметр конфигурации shared_buffers, 505
параметр конфигурации shared_ispell.max_size, 2567
параметр конфигурации shared_preload_libraries, 552
параметр конфигурации ssl, 502
параметр конфигурации ssl_ca_file, 502
параметр конфигурации ssl_cert_file, 502
параметр конфигурации ssl_ciphers, 503
параметр конфигурации ssl_crl_file, 503
параметр конфигурации ssl_dh_params_file, 504
параметр конфигурации ssl_ecdh_curve, 504
параметр конфигурации ssl_key_file, 503
параметр конфигурации ssl_prefer_server_ciphers, 503
параметр конфигурации standard_conforming_strings, 556
параметр конфигурации statement_timeout, 547
параметр конфигурации stats_temp_directory, 541
параметр конфигурации superuser_reserved_connections, 500
параметр конфигурации synchronize_seqscans, 556
параметр конфигурации synchronous_commit, 513
параметр конфигурации synchronous_standby_names, 520
параметр конфигурации syslog_facility, 532
параметр конфигурации syslog_ident, 533
параметр конфигурации syslog_sequence_numbers, 533
параметр конфигурации syslog_split_messages, 533
параметр конфигурации tcp_keepalives_count, 502
параметр конфигурации tcp_keepalives_idle, 501
параметр конфигурации tcp_keepalives_interval, 502
параметр конфигурации temp_buffers, 506
параметр конфигурации temp_file_limit, 508
параметр конфигурации temp_tablespaces, 545
параметр конфигурации TimeZone, 549
параметр конфигурации timezone_abbreviations, 550
параметр конфигурации trace_locks, 560
параметр конфигурации trace_lock_oidmin, 561
параметр конфигурации trace_lock_table, 561
параметр конфигурации trace_lwlocks, 560
параметр конфигурации trace_notify, 560
параметр конфигурации trace_recovery_messages, 560
параметр конфигурации trace_sort, 560
параметр конфигурации trace_userlocks, 561
параметр конфигурации track_activities, 541
параметр конфигурации track_activity_query_size, 541
параметр конфигурации track_commit_timestamp, 520
параметр конфигурации track_counts, 541
параметр конфигурации track_functions, 541
параметр конфигурации track_io_timing, 541
параметр конфигурации transaction_deferrable, 547
параметр конфигурации transaction_isolation, 547
параметр конфигурации transaction_read_only, 547
параметр конфигурации transform_null_equals, 556
параметр конфигурации unix_socket_directories, 500
параметр конфигурации unix_socket_group, 501
параметр конфигурации unix_socket_permissions, 501

параметр конфигурации update_process_title, 540
параметр конфигурации vacuum_cost_delay, 508
параметр конфигурации vacuum_cost_limit, 509
параметр конфигурации vacuum_cost_page_dirty, 509
параметр конфигурации vacuum_cost_page_hit, 509
параметр конфигурации vacuum_cost_page_miss, 509
параметр конфигурации vacuum_defer_cleanup_age, 521
параметр конфигурации vacuum_freeze_min_age, 548
параметр конфигурации vacuum_freeze_table_age, 548
параметр конфигурации vacuum_multixact_freeze_min_age, 548
параметр конфигурации vacuum_multixact_freeze_table_age, 548
параметр конфигурации wal_block_size, 558
параметр конфигурации wal_buffers, 516
параметр конфигурации wal_compression, 516
параметр конфигурации wal_consistency_checking, 561
параметр конфигурации wal_debug, 561
параметр конфигурации wal_keep_segments, 519
параметр конфигурации wal_level, 512
параметр конфигурации wal_log_hints, 515
параметр конфигурации wal_receiver_status_interval, 522
параметр конфигурации wal_receiver_timeout, 523
параметр конфигурации wal_retrieve_retry_interval, 523
параметр конфигурации wal_segment_size, 558
параметр конфигурации wal_sender_timeout, 520
параметр конфигурации wal_sync_method, 514
параметр конфигурации wal_writer_delay, 516
параметр конфигурации wal_writer_flush_after, 516
параметр конфигурации work_mem, 506
параметр конфигурации xmlbinary, 548
параметр конфигурации xmloption, 549
параметр конфигурации zero_damaged_pages, 562
параметры хранения, 1498
пароль, 584
  суперпользователя, 470
первичный ключ, 53
перегрузка
  операторов, 1034
  функций, 999
переиндексация, 620
перекрёстное соединение, 94
переменная окружения, 797
пересечение линии регрессии, 293
пересечение множеств, 109
переходные таблицы, 1519
  (см. также эфемерное именованное отношение)
  обращение из триггера на C, 1067
  реализация на языках программирования, 1245
план запроса, 430
подготовка запроса
  в PL/pgSQL, 1158
  в PL/Python, 1205
  в PL/Tcl, 1174
подготовленные операторы
  выполнение, 1599
  освобождение, 1542
  показ плана запроса, 1600
  создание, 1636
подзапрос, 11, 40, 98, 300
подмена сервера, 487
подтранзакции
  в PL/Tcl, 1179
поиск по шаблону, 208
поиск текста, 377
  функции и операторы, 147
поле
  вычисляемое, 171
полиморфная функция, 984
полиморфный тип, 984
политика, 61
полное имя, 67
полнотекстовый поиск, 377
  типы данных, 147
  функции и операторы, 147
получение параметров соединения через LDAP, 800
пользователь, 310, 583
  текущий, 310
пользователь postgres, 469
последовательное сканирование, 525
последовательность, 279
потоки
  с libpq, 805
права
  с правилами, 1103
  с представлениями, 1103
  проверка, 312
  для схем, 69
правила
  в сравнении с триггерами, 1105
  и представления, 1084
правила сортировки, 598
  в функциях SQL, 999
правило, 1082
  для DELETE, 1093
  для INSERT, 1093
  для SELECT, 1084
  для UPDATE, 1093
  и материализованные представления, 1090
правило сортировки

на PL/pgSQL, 1118
право, 60
право подключения, 584
правое соединение, 95
предикатная блокировка, 418
предложение OVER, 37
представление, 14
изменение, 1097
материализованное, 1090
реализация через правила, 1084
приведение
преобразование ввода/вывода, 1405
приведение типа, 29, 39
применимая роль, 925
приёмник замечаний, 790
провайдер нестандартного сканирования
обработчик, 2067
проверке подлинности
тайм-аут при, 502
производительность, 430
протокол
клиент-серверный, 1995
процедурный язык, 1108
обработчик, 2041
поддерживаемый отдельно, 2720
процентиль
дискретный, 296
непрерывный, 295
прямая, 142
прямоугольник, 143
псевдоним
в предложении FROM, 97
в списке выборки, 108
таблицы в запросе, 11
путь
для схем, 544
путь поиска, 68
видимость объектов, 315
текущий, 310
разделяемая библиотека, 1010
разделяемая память, 474
разрешение (см. право)
расширение, 1052
отдельно поддерживаемое, 2721
расширение SQL, 983
регламентное обслуживание, 611
регулярное выражение, 209, 210
(см. также поиск по шаблону)
регулярные выражения
и локали, 598
режим движущегося агрегата, 1024
резервная копия, 328, 623
резервный сервер, 640
рекомендательная блокировка, 425
реляционная база данных, 6
репликация, 640
ролей
право для создания, 584
роль, 583, 587
доступная, 947
право для запуска репликации, 584
право для игнорирования, 585
право для наследования, 584
право для ограничения соединений, 585
применимая, 925
членство, 585
секционирование, 75
секционирование данных, 640
секционированная таблица, 75
семафоры, 474
семейство операторов, 371, 1047
сетевые адреса
типы данных, 144
сигнал
серверные процессы, 328
символьная строка
конкатенация, 189
типы данных, 125
синтаксис
SQL, 23
синтаксис спецпоследовательностей, 25
синхронная фиксация данных, 725
системный каталог
схема, 70
скаляр (см. выражение)
сканирование по битовой карте, 367, 524
сканирование по индексу, 524
сканирование только индекса, 373
скобки, 32
следящий сервер, 640
слот репликации
логическая репликация, 1272
событийный триггер, 1073
в PL/Tcl, 1178
на C, 1078
соединение, 9, 94
внешнее, 10, 94
естественное, 95
замкнутое, 11
левое, 95
перекрёстное, 94
справа, 95
управление порядком, 444
сообщение об ошибке, 754
сопоставление пользователей, 86
сортировка, 109
сортирующая агрегатная функция, 35
сортирующие агрегатные функции
встроенные, 295
составной тип, 167, 983
константа, 167
конструктор, 42
сравнение, 303
спецсимвол обратная косая черта, 25
список отношений, 1082
сравнение

конструкторов строк, 303
операторы, 182
со строкой-результатом подзапроса, 300
составных типов, 303
сравнение табличных строк, 303
среднее, 290
средства администрирования
поддерживаемые отдельно, 2720
ссылка на столбец, 33
ссылочная целостность, 14, 54
стандартное отклонение, 294
по выборке, 294
по совокупности, 294
статистика, 293, 669
планировщика, 441, 442, 613
столбец, 6, 47
добавление, 58
переименование, 60
системный столбец, 57
удаление, 59
сторонние данные, 86
сторонняя таблица, 86
строка, 6, 47 (см. символьная строка)
строка символов
длина, 189
строки
апостроф с обратной косой, 554
предупреждение о спецсимволах, 555
соответствие стандарту, 556
суперпользователь, 4, 584
схема, 67, 590
public, 68
создание, 67
текущая, 69, 310
удаление, 68
таблица, 6, 47
изменение, 58
наследование, 71
переименование, 60
секционирование, 75
создание, 47
удаление, 48
табличная функция, 99
XMLTABLE, 263
табличное выражение, 93
табличное пространство, 593
временное, 545
по умолчанию, 545
тайм-аут
взаимоблокировка, 553
проверка подлинности клиента, 502
текстовая строка
константа, 25
текстовый поиск
индексы, 409
типы данных, 147
тип (см. тип данных)
полиморфный, 984
тип данных, 118
numeric, 120
базовый, 983
внутренняя организация, 1004
категория, 351
определённый пользователем, 1030
перечисление (enum), 140
преобразование, 350
приведение типа, 38
составной, 983
тип данных столбца
изменение, 60
тип табличной строки, 167
конструктор, 42
типы данных
константы, 29
типы перечислений, 140
точка, 142, 248
точка перезапуска, 728
точки монтирования файловых систем, 470
точки сохранения
определение, 1657
освобождение, 1647
откат, 1655
транзакция, 15
транзакция без записи
установка, 1692
транзакция в режиме «только чтение», 547
выбор по умолчанию, 546
трансляция журналов, 640
триггер
в PL/pgSQL, 1147
в PL/Python, 1203
для обновления столбца с производным значением tsvector, 392
на языке PL/Tcl, 1176
триггеры
в сравнении с правилами, 1105
тёплый резерв, 640
угроза стабильности, 449
удаление, 91
управляющий файл, 1053
уровень изоляции транзакции, 547
выбор по умолчанию, 546
установка, 1692
уровень изоляции транзакций, 415
read committed, 415
repeatable read, 417
serializable, 418
условное выражение, 281
утверждения
в PL/pgSQL, 1147
файл паролей, 798
файл соединений служб, 799
фантомное чтение, 415
форматирование, 224
функции
пользовательские

на SQL, 985
разрешение типов при вызове, 355
функции, возвращающие множества
функции, 306
функциональная зависимость, 104
функция, 182
RETURNS TABLE, 997
в предложении FROM, 99
внутренняя, 1002
вызов, 34
выходной параметр, 990
значения аргументов по умолчанию, 992
именная передача, 45
именованный аргумент, 986
определяемая пользователем, 985
переменные параметры, 991
позиционная запись, 45
полиморфная, 984
пользовательская
на C, 1002
с SETOF, 993
смешанная запись, 46
функция instr, 1168
функция ввода, 1030
функция вывода, 1030
функция завершения работы библиотеки, 1003
функция инициализации библиотеки, 1003
функция с переменными параметрами, 991
хеш (см. индекс)
цикл
в PL/pgSQL, 1132
часовой пояс, 135, 549
ввод аббревиатур, 2162
преобразование, 241
указание в стиле POSIX, 2163
числа с произвольной точностью, 120
число с плавающей точкой
отображение, 550
числовые
константы, 28
чувствительность к регистру
в командах SQL, 24
шаблоны
в psql и pg_dump, 1818
шифрование, 488
избранных столбцов, 2484
экранированные строки
в libpq, 770
экспорт в XML, 266
эфемерное именованное отношение
разрегистрация в SPI, 1244
регистрация в SPI, 1243, 1245

A

abbrev, 250
ABORT, 1282
abs, 186
acos, 188
acosd, 188
adminpack, 2382
age, 232
AIX
настройка IPC, 476
akeys, 2448
ALL, 300, 303
ALTER AGGREGATE, 1283
ALTER COLLATION, 1285
ALTER CONVERSION, 1287
ALTER DATABASE, 1288
ALTER DEFAULT PRIVILEGES, 1290
ALTER DOMAIN, 1293
ALTER EVENT TRIGGER, 1296
ALTER EXTENSION, 1297
ALTER FOREIGN DATA WRAPPER, 1300
ALTER FOREIGN TABLE, 1302
ALTER FUNCTION, 1307
ALTER GROUP, 1310
ALTER INDEX, 1311
ALTER LANGUAGE, 1313
ALTER LARGE OBJECT, 1314
ALTER MATERIALIZED VIEW, 1315
ALTER OPERATOR, 1317
ALTER OPERATOR CLASS, 1319
ALTER OPERATOR FAMILY, 1320
ALTER POLICY, 1324
ALTER PUBLICATION, 1325
ALTER ROLE, 585, 1327
ALTER RULE, 1331
ALTER SCHEMA, 1332
ALTER SEQUENCE, 1333
ALTER SERVER, 1336
ALTER STATISTICS, 1338
ALTER SUBSCRIPTION, 1339
ALTER SYSTEM, 1341
ALTER TABLE, 1343
ALTER TABLESPACE, 1357
ALTER TEXT SEARCH CONFIGURATION, 1358
ALTER TEXT SEARCH DICTIONARY, 1360
ALTER TEXT SEARCH PARSER, 1362
ALTER TEXT SEARCH TEMPLATE, 1363
ALTER TRIGGER, 1364
ALTER TYPE, 1365
ALTER USER, 1368
ALTER USER MAPPING, 1369
ALTER VIEW, 1370
amcheck, 2384
ANALYZE, 613, 1372
AND (оператор), 182
any, 179
ANY, 292, 300, 303
anyarray, 179
anyarray_elemtype, 2437
anyarray_to_text, 2436
anyelement, 179
anyenum, 179
anynonarray, 179

anyrange, 179
area, 246
armor, 2489
ARRAY, 40
  определение типа результата, 359
array_agg, 290, 2454
array_append, 285
array_cat, 285
array_dims, 285
array_fill, 285
array_length, 285
array_lower, 285
array_ndims, 285
array_position, 285
array_positions, 285
array_prepend, 285
array_remove, 285
array_replace, 285
array_to_json, 272
array_to_string, 285
array_to_tsvector, 252
array_upper, 285
ascii, 191
asin, 188
asind, 188
ASSERT
  в PL/pgSQL, 1147
AT TIME ZONE, 241
atan, 188
atan2, 188
atan2d, 188
atand, 188
auth_delay, 2387
auto-increment (см. serial)
autocommit
  psql, 1819
auto_explain, 2387
avals, 2448
avg, 290

B

B-дерево (см. индекс)
BASE_BACKUP, 2015
BEGIN, 1375
BETWEEN, 183
BETWEEN SYMMETRIC, 184
BGWORKER_BACKEND_DATABASE_CONNECTION, 1268
BGWORKER_SHMEM_ACCESS, 1267
bigint, 28, 120
bigserial, 123
bit_and, 290
bit_length, 189
bit_or, 290
BLOB (см. большой объект)
bloom, 2390
bool_and, 291
bool_or, 291
box, 247
box (тип данных), 143
BRIN (см. индекс)
brin_desummarize_range, 341
brin_metapage_info, 2479
brin_page_items, 2480
brin_page_type, 2479
brin_revmap_data, 2480
brin_summarize_new_values, 341
brin_summarize_range, 341
broadcast, 250
btree_gin, 2393
btree_gist, 2393
btrim, 191, 206
bt_index_check, 2384
bt_index_parent_check, 2385
bt_metap, 2478
bt_page_items, 2478, 2479
bt_page_stats, 2478
bytea, 127

C

C, 738, 828
C++, 1022
cardinality, 285
CASCADE
  с DROP, 87
  действие внешнего ключа, 55
CASE, 281
  определение типа результата, 359
cbrt, 186
ceil, 186
ceiling, 186
center, 246
Certificate, 580
char, 125
character, 125
character varying, 125
char_length, 189
CHECK OPTION, 1537
CHECKPOINT, 1377
chkpass, 2394
chr, 191
cid, 178
cidr, 144
citext, 2395
clock_timestamp, 232
CLOSE, 1378
CLUSTER, 1379
clusterdb, 1714
cmax, 57
cmin, 57
COALESCE, 283
COLLATE, 39
collation for, 316
col_description, 322
COMMENT, 1381
COMMIT, 1385

COMMIT PREPARED, 1386
concat, 191
concat_ws, 191
connectby, 2573, 2579
conninfo, 745
CONTINUE
  в PL/pgSQL, 1133
convert, 192
convert_from, 192
convert_to, 192
COPY, 8, 1387
  с libpq, 781
corr, 293
cos, 188
cosd, 188
cot, 188
cotd, 188
count, 291
covar_pop, 293
covar_samp, 293
CREATE ACCESS METHOD, 1397
CREATE AGGREGATE, 1398
CREATE CAST, 1405
CREATE COLLATION, 1409
CREATE CONVERSION, 1411
CREATE DATABASE, 590, 1413
CREATE DOMAIN, 1416
CREATE EVENT TRIGGER, 1419
CREATE EXTENSION, 1421
CREATE FOREIGN DATA WRAPPER, 1424
CREATE FOREIGN TABLE, 1426
CREATE FUNCTION, 1430
CREATE GROUP, 1438
CREATE INDEX, 1439
CREATE LANGUAGE, 1446
CREATE MATERIALIZED VIEW, 1449
CREATE OPERATOR, 1451
CREATE OPERATOR CLASS, 1454
CREATE OPERATOR FAMILY, 1457
CREATE POLICY, 1458
CREATE PUBLICATION, 1464
CREATE ROLE, 583, 1466
CREATE RULE, 1471
CREATE SCHEMA, 1474
CREATE SEQUENCE, 1476
CREATE SERVER, 1480
CREATE STATISTICS, 1482
CREATE SUBSCRIPTION, 1484
CREATE TABLE, 6, 1487
CREATE TABLE AS, 1506
CREATE TABLESPACE, 594, 1509
CREATE TEXT SEARCH CONFIGURATION, 1511
CREATE TEXT SEARCH DICTIONARY, 1512
CREATE TEXT SEARCH PARSER, 1514
CREATE TEXT SEARCH TEMPLATE, 1516
CREATE TRANSFORM, 1517
CREATE TRIGGER, 1519
CREATE TYPE, 1526
CREATE USER, 1535
CREATE USER MAPPING, 1536
CREATE VIEW, 1537
createdb, 2, 591, 1717
createuser, 583, 1720
CREATE_REPLICATION_SLOT, 2011
crosstab, 2574, 2576, 2577
crypt, 2485
cstring, 179
ctid, 57
CTID, 1089
CUBE, 104
cube (расширение), 2400
cume_dist, 298
  гипотезирующая функция, 297
current_catalog, 310
current_database, 310
current_date, 233
current_logfiles
  и функция pg_current_logfile, 311
current_query, 310
current_role, 310
current_schema, 310
current_schemas, 310
current_setting, 327
current_time, 233
current_timestamp, 233
current_user, 310
currval, 279

D

date, 129, 131
date_part, 233, 237
date_trunc, 233, 240
dblink, 2405, 2411
dblink_build_sql_delete, 2432
dblink_build_sql_insert, 2430
dblink_build_sql_update, 2433
dblink_cancel_query, 2428
dblink_close, 2420
dblink_connect, 2406
dblink_connect_u, 2409
dblink_disconnect, 2410
dblink_error_message, 2422
dblink_exec, 2414
dblink_fetch, 2418
dblink_get_connections, 2421
dblink_get_notify, 2425
dblink_get_pkey, 2429
dblink_get_result, 2426
dblink_is_busy, 2424
dblink_open, 2416
dblink_send_query, 2423
DEALLOCATE, 1542
dearmor, 2489
decimal (см. numeric)
DECLARE, 1543
decode, 192, 206

decode_bytea
  в PL/Perl, 1189
decrypt, 2492
decrypt_iv, 2492
defined, 2449
degrees, 186
DELETE, 13, 91, 1547
  RETURNING, 91
delete, 2450
dense_rank, 298
  гипотезирующая функция, 297
diameter, 246
dict_int, 2434
dict_xsyn, 2434
difference, 2444
digest, 2484
DISCARD, 1550
DISTINCT, 9, 108
div, 186
dmetaphone, 2446
dmetaphone_alt, 2446
DO, 1551
double precision, 122
DROP ACCESS METHOD, 1552
DROP AGGREGATE, 1553
DROP CAST, 1555
DROP COLLATION, 1556
DROP CONVERSION, 1557
DROP DATABASE, 593, 1558
DROP DOMAIN, 1559
DROP EVENT TRIGGER, 1560
DROP EXTENSION, 1561
DROP FOREIGN DATA WRAPPER, 1562
DROP FOREIGN TABLE, 1563
DROP FUNCTION, 1564
DROP GROUP, 1566
DROP INDEX, 1567
DROP LANGUAGE, 1569
DROP MATERIALIZED VIEW, 1570
DROP OPERATOR, 1571
DROP OPERATOR CLASS, 1573
DROP OPERATOR FAMILY, 1575
DROP OWNED, 1576
DROP POLICY, 1577
DROP PUBLICATION, 1578
DROP ROLE, 583, 1579
DROP RULE, 1580
DROP SCHEMA, 1581
DROP SEQUENCE, 1582
DROP SERVER, 1583
DROP STATISTICS, 1584
DROP SUBSCRIPTION, 1585
DROP TABLE, 7, 1586
DROP TABLESPACE, 1587
DROP TEXT SEARCH CONFIGURATION, 1588
DROP TEXT SEARCH DICTIONARY, 1589
DROP TEXT SEARCH PARSER, 1590
DROP TEXT SEARCH TEMPLATE, 1591
DROP TRANSFORM, 1592
DROP TRIGGER, 1593
DROP TYPE, 1594
DROP USER, 1595
DROP USER MAPPING, 1596
DROP VIEW, 1597
dropdb, 593, 1724
dropuser, 583, 1726
DROP_REPLICATION_SLOT, 2014
DTD, 150
DTrace, 708
dump_stat, 2436
dump_statistic, 2436
dynamic_library_path, 1003

E

each, 2449
earth, 2439
earthdistance, 2438
earth_box, 2439
earth_distance, 2439
ECPG, 828
ecpg, 1728
elog
  в PL/Perl, 1189
  в PL/Python, 1209
  в PL/Tcl, 1176
encode, 192, 206
encode_array_constructor
  в PL/Perl, 1190
encode_array_literal
  в PL/Perl, 1190
encode_bytea
  в PL/Perl, 1189
encode_typed_literal
  в PL/Perl, 1190
encrypt, 2492
encrypt_iv, 2492
END, 1598
enum_first, 244
enum_last, 244
enum_range, 244
event_trigger, 179
every, 291
EXCEPT, 109
EXECUTE, 1599
exist, 2449
EXISTS, 300
EXIT
  в PL/pgSQL, 1132
exp, 186
EXPLAIN, 430, 1600
extract, 233, 237

F

factorial, 186
false, 139
family, 250

fdw_handler, 179
FETCH, 1605
file_fdw, 2440
FILTER, 35
first_value, 299
float4 (см. real)
float8 (см. double precision)
floating point, 122
floor, 186
format, 193, 203
  использование в PL/pgSQL, 1124
format_type, 316
FreeBSD
  настройка IPC, 477
  разделяемая библиотека, 1011
  скрипт запуска, 472
FSM (см. Карта свободного пространства)
fsm_page_contents, 2477
fuzzystrmatch, 2443

G

gc_to_sec, 2439
generate_series, 306
generate_subscripts, 307
gen_random_bytes, 2493
gen_random_uuid, 2493
gen_salt, 2486
GEQO (см. генетическая оптимизация запросов)
get_bit, 206
get_byte, 206
get_current_ts_config, 252
get_namespace, 2438
get_raw_page, 2476
GIN (см. индекс)
gin_clean_pending_list, 341
gin_leafpage_items, 2481
gin_metapage_info, 2480
gin_page_opaque_info, 2480
GiST (см. индекс)
GRANT, 60, 1609
GREATEST, 284
  определение типа результата, 359
GROUP BY, 12, 102
GROUPING, 297
GROUPING SETS, 104
GSSAPI, 574
GUID, 149

H

hash_bitmap_info, 2482
hash_metapage_info, 2482
hash_page_items, 2481
hash_page_stats, 2481
hash_page_type, 2481
has_any_column_privilege, 314
has_column_privilege, 314
has_database_privilege, 314
has_foreign_data_wrapper_privilege, 314
has_function_privilege, 314
has_language_privilege, 314
has_schema_privilege, 314
has_sequence_privilege, 314
has_server_privilege, 314
has_tablespace_privilege, 314
has_table_privilege, 314
has_type_privilege, 314
HAVING, 12, 104
heap_page_items, 2477
heap_page_item_attrs, 2478
height, 246
hmac, 2485
host, 250
hostmask, 250
HP-UX
  настройка IPC, 478
  разделяемая библиотека, 1011
hstore, 2446, 2448
hstore_to_array, 2448
hstore_to_json, 2449
hstore_to_jsonb, 2449
hstore_to_jsonb_loose, 2449
hstore_to_json_loose, 2449
hstore_to_matrix, 2448

I

icount, 2455
ICU, 600, 1409
ident, 576
IDENTIFY_SYSTEM, 2010
idx, 2456
IFNULL, 283
IMMUTABLE, 1000
IMPORT FOREIGN SCHEMA, 1616
IN, 300, 303
include_dir
  в файле конфигурации, 498
include_if_exists
  в файле конфигурации, 498
index_am_handler, 179
inet (тип данных), 144
inet_client_addr, 311
inet_client_port, 311
inet_merge, 250
inet_same_family, 250
inet_server_addr, 311
inet_server_port, 311
initcap, 193
initdb, 469, 1839
INSERT, 7, 89, 1618
  RETURNING, 91
int2 (см. smallint)
int4 (см. integer)
int8 (см. bigint)
intagg, 2454
intarray, 2455
integer, 28, 120

internal, 179
INTERSECT, 109
interval, 129, 136
intset, 2456
int_array_aggregate, 2454
int_array_enum, 2454
IS DISTINCT FROM, 184, 303
IS DOCUMENT, 261
IS FALSE, 184
IS NOT DISTINCT FROM, 184, 303
IS NOT DOCUMENT, 261
IS NOT FALSE, 184
IS NOT NULL, 184
IS NOT TRUE, 184
IS NOT UNKNOWN, 184
IS NULL, 184, 556
IS TRUE, 184
IS UNKNOWN, 184
isclosed, 246
isempty, 289
isfinite, 233
isn, 2458
ISNULL, 184
isn_weak, 2459
isopen, 246
is_array_ref
  в PL/Perl, 1190
is_valid, 2460

J

JSON, 152
  функции и операторы, 269
JSONB, 152
jsonb
  вхождение, 154
  индексы по, 156
  существование, 154
jsonb_agg, 291
jsonb_array_elements, 273
jsonb_array_elements_text, 273
jsonb_array_length, 273
jsonb_build_array, 272
jsonb_build_object, 272
jsonb_each, 273
jsonb_each_text, 273
jsonb_extract_path, 273
jsonb_extract_path_text, 273
jsonb_insert, 273
jsonb_object, 272
jsonb_object_agg, 291
jsonb_object_keys, 273
jsonb_populate_record, 273
jsonb_populate_recordset, 273
jsonb_pretty, 273
jsonb_set, 273
jsonb_strip_nulls, 273
jsonb_to_record, 273
jsonb_to_recordset, 273
jsonb_typeof, 273
json_agg, 291
json_array_elements, 273
json_array_elements_text, 273
json_array_length, 273
json_build_array, 272
json_build_object, 272
json_each, 273
json_each_text, 273
json_extract_path, 273
json_extract_path_text, 273
json_object, 272
json_object_agg, 291
json_object_keys, 273
json_populate_record, 273
json_populate_recordset, 273
json_strip_nulls, 273
json_to_record, 273
json_to_recordset, 273
json_typeof, 273
justify_days, 234
justify_hours, 234
justify_interval, 234

L

lag, 299
language_handler, 179
lastval, 279
last_value, 299
LATERAL
  в предложении FROM, 100
latitude, 2439
lca, 2471
LDAP, 577
lead, 299
LEAST, 284
  определение типа результата, 359
left, 193
length, 193, 207, 246, 252
length(tsvector), 389
levenshtein, 2444
levenshtein_less_equal, 2444
libpq, 738
  однострочный режим, 777
libpq-fe.h, 738, 751
libpq-int.h, 751
LIKE, 208
  и локали, 598
LIMIT, 110
Linux
  настройка IPC, 478
  разделяемая библиотека, 1011
  скрипт запуска, 472
LISTEN, 1625
ll_to_earth, 2439
ln, 186
lo, 2466
LOAD, 1627

localtime, 234
localtimestamp, 234
lock, 420
LOCK, 421, 1628
log, 186
longitude, 2439
looks_like_number
  в PL/Perl, 1190
lower, 189, 289
  и локали, 597
lower_inc, 289
lower_inf, 289
lo_close, 821
lo_creat, 818, 821
lo_create, 818
lo_export, 819, 821
lo_from_bytea, 821
lo_get, 821
lo_import, 818, 821
lo_import_with_oid, 818
lo_lseek, 820
lo_lseek64, 820
lo_open, 819
lo_put, 821
lo_read, 819
lo_tell, 820
lo_tell64, 820
lo_truncate, 820
lo_truncate64, 820
lo_unlink, 821, 821
lo_write, 819
lpad, 193
lseg, 142, 248
LSN, 729
ltree, 2467
ltree2text, 2471
ltrim, 194

M

MAC-адрес (см. macaddr)
MAC-адрес (в формате EUI-64) (см. macaddr)
macaddr (тип данных), 145
macaddr8 (тип данных), 145
macaddr8_set7bit, 251
macOS
  настройка IPC, 478
  разделяемая библиотека, 1011
make_date, 234
make_interval, 234
make_time, 234
make_timestamp, 235
make_timestamptz, 235
make_valid, 2459
mamonsu, 2662
masklen, 250
max, 291
md5, 194, 207
MD5, 573
memory overcommit, 482
metaphone, 2445
min, 292
mod, 186
MOVE, 1631
MultiXactId, 618
MVCC, 414

N

NaN (см. не число)
NetBSD
  настройка IPC, 477
  разделяемая библиотека, 1011
  скрипт запуска, 473
netmask, 250
network, 250
nextval, 279
NFS (см. Сетевые файловые системы)
nlevel, 2471
normal_rand, 2573
NOT (оператор), 182
NOT IN, 300, 303
NOTIFY, 1633
  в libpq, 780
NOTNULL, 184
now, 235
npoints, 246
nth_value, 299
ntile, 299
NULL
  сравнение, 184
NULL-значение
  в DISTINCT, 108
  в PL/Perl, 1183
  в PL/Python, 1199
NULLIF, 283
numeric, 28
numeric (тип данных), 120
numnode, 252, 390
num_nonnulls, 185
num_nulls, 185
NVL, 283

O

obj_description, 322
octet_length, 189, 205
OFFSET, 110
OID
  в libpq, 769
  столбец, 57
oid, 178
oid2name, 2594
ON CONFLICT, 1618
ONLY, 94
OOM, 482
opaque, 179
OpenBSD
  настройка IPC, 477

разделяемая библиотека, 1012
скрипт запуска, 472
OR (оператор), 182
Oracle
портирование из PL/SQL в PL/pgSQL, 1161
ORDER BY, 9, 109
и локали, 597
ordinality, 308
overcommit, 482
OVERLAPS, 235
overlay, 190, 205

P

pageinspect, 2476
page_checksum, 2477
page_header, 2476
palloc, 1010
PAM, 581
parse_ident, 194
password
Аутентификация, 573
passwordcheck, 2482
path, 248
path (тип данных), 143
pclose, 246
peer, 577
percent_rank, 298
гипотезирующая функция, 297
Perl, 1182
pfree, 1010
pg-setup, 1858
pg-wrapper, 1790
PGAPPNAME, 798
pgbench, 1737
pgbouncer, 2681
PGcancel, 778
PGCLIENTENCODING, 798
PGconn, 738
PGCONNECT_TIMEOUT, 798
pgcrypto, 2484
PGDATA, 469
PGDATABASE, 797
PGDATESTYLE, 798
PGEventProc, 793
PGGEQO, 798
PGGSSLIB, 798
PGHOST, 797
PGHOSTADDR, 797
PGKRBSRVNAME, 798
PGLOCALEDIR, 798
PGOPTIONS, 798
PGPASSFILE, 797
PGPASSWORD, 797
PGPORT, 797
pgpro_controldata, 2714
pgpro_edition, 312
pgpro_version, 312
pgp_armor_headers, 2489
pgp_key_id, 2489
pgp_pub_decrypt, 2488
pgp_pub_decrypt_bytea, 2488
pgp_pub_encrypt, 2488
pgp_pub_encrypt_bytea, 2488
pgp_sym_decrypt, 2488
pgp_sym_decrypt_bytea, 2488
pgp_sym_encrypt, 2488
pgp_sym_encrypt_bytea, 2488
PGREQUIREPEER, 798
PGREQUIRESSL, 798
PGresult, 761
pgrowlocks, 2522, 2523
PGSERVICE, 797
PGSERVICEFILE, 798
PGSSLCERT, 798
PGSSLCOMPRESSION, 798
PGSSLCRL, 798
PGSSLKEY, 798
PGSSLMODE, 798
PGSSLROOTCERT, 798
pgstatginindex, 2531
pgstathashindex, 2531
pgstatindex, 2530
pgstattuple, 2529, 2529
pgstattuple_approx, 2532
PGSYSCONFDIR, 798
PGTARGETSESSIONATTRS, 798
PGTZ, 798
PGUSER, 797
pgxs, 1061
pg_advisory_lock, 344
pg_advisory_lock_shared, 345
pg_advisory_unlock, 345
pg_advisory_unlock_all, 345
pg_advisory_unlock_shared, 345
pg_advisory_xact_lock, 345
pg_advisory_xact_lock_shared, 345
pg_aggregate, 1891
pg_am, 1893
pg_amop, 1894
pg_amproc, 1895
pg_archivecleanup, 1843
pg_attrdef, 1896
pg_attribute, 1896
pg_authid, 1900
pg_auth_members, 1901
pg_available_extensions, 1967
pg_available_extension_versions, 1968
pg_backend_pid, 310
pg_backup_start_time, 328
pg_basebackup, 1730
pg_blocking_pids, 311
pg_buffercache, 2483
pg_buffercache_pages, 2483
pg_cancel_backend, 327
pg_cast, 1901
pg_class, 1903

pg_client_encoding, 195
pg_collation, 1908
pg_collation_actual_version, 341
pg_collation_is_visible, 316
pg_column_size, 338
pg_config, 1750, 1968
  c ecpg, 881
  c libpq, 806
  для пользовательских функций на C, 1010
pg_conf_load_time, 311
pg_constraint, 1909
pg_controldata, 1845
pg_control_checkpoint, 325
pg_control_init, 325
pg_control_recovery, 325
pg_control_system, 325
pg_conversion, 1912
pg_conversion_is_visible, 316
pg_create_logical_replication_slot, 335
pg_create_physical_replication_slot, 334
pg_create_restore_point, 328
pg_ctl, 469, 471, 1846
pg_current_logfile, 311
pg_current_wal_flush_lsn, 328
pg_current_wal_insert_lsn, 328
pg_current_wal_lsn, 328
pg_cursors, 1968
pg_database, 592, 1913
pg_database_size, 338
pg_db_role_setting, 1915
pg_ddl_command, 179
pg_default_acl, 1916
pg_depend, 1917
pg_describe_object, 321
pg_description, 1918
pg_drop_replication_slot, 334
pg_dump, 1753
pg_dumpall, 1765
  использование при обновлении, 486
pg_enum, 1919
pg_event_trigger, 1919
pg_event_trigger_ddl_commands, 346
pg_event_trigger_dropped_objects, 347
pg_event_trigger_table_rewrite_oid, 349
pg_event_trigger_table_rewrite_reason, 349
pg_export_snapshot, 333
pg_extension, 1920
pg_extension_config_dump, 1056
pg_filedump, 2711
pg_filenode_relation, 340
pg_file_rename, 2383
pg_file_settings, 1969
pg_file_unlink, 2383
pg_file_write, 2383
pg_foreign_data_wrapper, 1921
pg_foreign_server, 1922
pg_foreign_table, 1922
pg_freespace, 2495
pg_freespacemap, 2495
pg_function_is_visible, 316
pg_get_constraintdef, 316
pg_get_expr, 316
pg_get_functiondef, 316
pg_get_function_arguments, 316
pg_get_function_identity_arguments, 316
pg_get_function_result, 316
pg_get_indexdef, 316
pg_get_keywords, 316
pg_get_object_address, 321
pg_get_ruledef, 316
pg_get_serial_sequence, 316
pg_get_statisticsobjdef, 316
pg_get_triggerdef, 316
pg_get_userbyid, 316
pg_get_viewdef, 316
pg_group, 1970
pg_has_role, 314
pg_hba.conf, 564
pg_hba_file_rules, 1970
pg_ident.conf, 571
pg_identify_object, 321
pg_identify_object_as_address, 321
pg_import_system_collations, 341
pg_index, 1923
pg_indexam_has_property, 316
pg_indexes, 1971
pg_indexes_size, 338
pg_index_column_has_property, 316
pg_index_has_property, 316
pg_inherits, 1926
pg_init_privs, 1926
pg_isready, 1771
pg_is_in_backup, 328
pg_is_in_recovery, 331
pg_is_other_temp_schema, 311
pg_is_wal_replay_paused, 332
pg_language, 1927
pg_largeobject, 1929
pg_largeobject_metadata, 1929
pg_last_committed_xact, 325
pg_last_wal_receive_lsn, 331
pg_last_wal_replay_lsn, 331
pg_last_xact_replay_timestamp, 331
pg_listening_channels, 312
pg_locks, 1972
pg_logdir_ls, 2383
pg_logical_emit_message, 337
pg_logical_slot_get_binary_changes, 336
pg_logical_slot_get_changes, 335
pg_logical_slot_peek_binary_changes, 336
pg_logical_slot_peek_changes, 335
pg_lsn, 179
pg_ls_dir, 343
pg_ls_logdir, 343
pg_ls_waldir, 343
pg_matviews, 1976

pg_my_temp_schema, 311
pg_namespace, 1930
pg_notification_queue_usage, 312
pg_notify, 1634
pg_opclass, 1930
pg_opclass_is_visible, 316
pg_operator, 1931
pg_operator_is_visible, 316
pg_opfamily, 1932
pg_opfamily_is_visible, 316
pg_options_to_table, 316
pg_partitioned_table, 1932
pg_pltemplate, 1934
pg_policies, 1976
pg_policy, 1935
pg_postmaster_start_time, 312
pg_prepared_statements, 1977
pg_prepared_xacts, 1978
pg_prewarm, 2516
pg_probackup, 2598
pg_proc, 1935
pg_publication, 1941
pg_publication_rel, 1941
pg_publication_tables, 1979
pg_range, 1941
pg_read_binary_file, 343
pg_read_file, 343
pg_receivewal, 1773
pg_receivexlog, 2744 (см. pg_receivewal)
pg_recvlogical, 1777
pg_relation_filenode, 340
pg_relation_filepath, 340
pg_relation_size, 338
pg_reload_conf, 328
pg_relpages, 2532
pg_repack, 2654
pg_replication_origin, 1942
pg_replication_origin_advance, 337
pg_replication_origin_create, 336
pg_replication_origin_drop, 336
pg_replication_origin_oid, 336
pg_replication_origin_progress, 337
pg_replication_origin_session_is_setup, 336
pg_replication_origin_session_progress, 336
pg_replication_origin_session_reset, 336
pg_replication_origin_session_setup, 336
pg_replication_origin_status, 1979
pg_replication_origin_xact_reset, 337
pg_replication_origin_xact_setup, 337
pg_replication_slots, 1979
pg_resetwal, 1852
pg_resetxlog, 2744 (см. pg_resetwal)
pg_restore, 1781
pg_rewind, 1855
pg_rewrite, 1942
pg_roles, 1981
pg_rotate_logfile, 328
pg_rules, 1983
pg_safe_snapshot_blocking_pids, 312
pg_seclabel, 1943
pg_seclabels, 1983
pg_sequence, 1944
pg_sequences, 1984
pg_service.conf, 799
pg_settings, 1985
pg_shadow, 1987
pg_shdepend, 1945
pg_shdescription, 1946
pg_shseclabel, 1946
pg_size_bytes, 338
pg_size_pretty, 338
pg_sleep, 243
pg_sleep_for, 243
pg_sleep_until, 243
pg_snapshot_any, 345
pg_standby, 2716
pg_start_backup, 328
pg_statio_all_indexes, 672
pg_statio_all_sequences, 672
pg_statio_all_tables, 672
pg_statio_sys_indexes, 672
pg_statio_sys_sequences, 673
pg_statio_sys_tables, 672
pg_statio_user_indexes, 672
pg_statio_user_sequences, 673
pg_statio_user_tables, 672
pg_statistic, 441, 1947
pg_statistics_obj_is_visible, 316
pg_statistic_ext, 442, 1949
pg_stats, 441, 1988
pg_stat_activity, 670
pg_stat_all_indexes, 672
pg_stat_all_tables, 671
pg_stat_archiver, 671
pg_stat_bgwriter, 671
pg_stat_clear_snapshot, 704
pg_stat_database, 671
pg_stat_database_conflicts, 671
pg_stat_file, 343
pg_stat_get_activity, 703
pg_stat_get_snapshot_timestamp, 704
pg_stat_progress_vacuum, 671
pg_stat_replication, 671
pg_stat_reset, 704
pg_stat_reset_shared, 704
pg_stat_reset_single_function_counters, 704
pg_stat_reset_single_table_counters, 704
pg_stat_ssl, 671
pg_stat_statements, 2524
  функция, 2527
pg_stat_statements_reset, 2527
pg_stat_subscription, 671
pg_stat_sys_indexes, 672
pg_stat_sys_tables, 671
pg_stat_user_functions, 673
pg_stat_user_indexes, 672

pg_stat_user_tables, 672
pg_stat_wal_receiver, 671
pg_stat_xact_all_tables, 672
pg_stat_xact_sys_tables, 672
pg_stat_xact_user_functions, 673
pg_stat_xact_user_tables, 672
pg_stop_backup, 328
pg_subscription, 1950
pg_subscription_rel, 1951
pg_switch_wal, 328
pg_tables, 1991
pg_tablespace, 1951
pg_tablespace_databases, 316
pg_tablespace_location, 316
pg_tablespace_size, 338
pg_table_is_visible, 316
pg_table_size, 338
pg_temp, 544
  защищённые функции, 1437
pg_terminate_backend, 328
pg_test_fsync, 1860
pg_test_timing, 1861
pg_timezone_abbrevs, 1991
pg_timezone_names, 1992
pg_total_relation_size, 338
pg_transform, 1952
pg_trgm, 2533
pg_trigger, 1953
pg_try_advisory_lock, 345
pg_try_advisory_lock_shared, 345
pg_try_advisory_xact_lock, 345
pg_try_advisory_xact_lock_shared, 345
pg_ts_config, 1955
pg_ts_config_is_visible, 316
pg_ts_config_map, 1955
pg_ts_dict, 1956
pg_ts_dict_is_visible, 316
pg_ts_parser, 1956
pg_ts_parser_is_visible, 316
pg_ts_template, 1957
pg_ts_template_is_visible, 316
pg_type, 1958
pg_typeof, 316
pg_type_is_visible, 316
pg_upgrade, 1865
pg_user, 1992
pg_user_mapping, 1966
pg_user_mappings, 1993
pg_views, 1994
pg_visibility, 2547
pg_waldump, 1873
pg_walfile_name, 328
pg_walfile_name_offset, 328
pg_wal_lsn_diff, 328
pg_wal_replay_pause, 332
pg_wal_replay_resume, 332
pg_xact_commit_timestamp, 325
pg_xlogdump, 2744 (см. pg_waldump)
phraseto_tsquery, 253, 385
pi, 187
PIC, 1011
PID
  определение PID серверного процесса
    в libpq, 755
PITR, 623
PITR резерв, 640
pkg-config
  с ecpg, 881
  с libpq, 806
PL/Perl, 1182
PL/PerlU, 1191
PL/pgSQL, 1111
PL/Python, 1196
PL/SQL (Oracle)
  портирование в PL/pgSQL, 1161
PL/Tcl, 1171
plainto_tsquery, 253, 385
popen, 246
populate_record, 2450
port, 747
position, 190, 205
POSTGRES, xxiii
postgres, 2, 471, 591, 1875
Postgres95, xxiii
postgresql.auto.conf, 496
postgresql.conf, 495
postgres_fdw, 2550
postmaster, 1882
power, 187
PQbackendPID, 755
PQbinaryTuples, 767
  с COPY, 781
PQcancel, 778
PQclear, 765
PQclientEncoding, 785
PQcmdStatus, 769
PQcmdTuples, 769
PQconndefaults, 742
PQconnectdb, 739
PQconnectdbParams, 738
PQconnectionNeedsPassword, 755
PQconnectionUsedPassword, 755
PQconnectPoll, 740
PQconnectStart, 740
PQconnectStartParams, 740
PQconninfo, 743
PQconninfoFree, 787
PQconninfoParse, 743
PQconsumeInput, 775
PQcopyResult, 788
PQdb, 752
PQdescribePortal, 760
PQdescribePrepared, 760
PQencryptPassword, 788
PQencryptPasswordConn, 787
PQendcopy, 785

PQerrorMessage, 754
PQescapeBytea, 772
PQescapeByteaConn, 772
PQescapeIdentifier, 770
PQescapeLiteral, 770
PQescapeString, 771
PQescapeStringConn, 771
PQexec, 757
PQexecParams, 757
PQexecPrepared, 760
PQfformat, 766
  с COPY, 782
PQfinish, 743
PQfireResultCreateEvents, 788
PQflush, 777
PQfmod, 767
PQfn, 779
PQfname, 765
PQfnumber, 765
PQfreeCancel, 778
PQfreemem, 787
PQfsize, 767
PQftable, 766
PQftablecol, 766
PQftype, 766
PQgetCancel, 778
PQgetCopyData, 783
PQgetisnull, 768
PQgetlength, 768
PQgetline, 783
PQgetlineAsync, 784
PQgetResult, 775
PQgetssl, 757
PQgetvalue, 767
PQhost, 752
PQinitOpenSSL, 804
PQinitSSL, 805
PQinstanceData, 794
PQisBusy, 776
PQisnonblocking, 777
PQisthreadsafe, 805
PQlibVersion, 789
  (см. также PQserverVersion)
PQmakeEmptyPGresult, 788
PQnfields, 765
  с COPY, 781
PQnotifies, 780
PQnparams, 768
PQntuples, 765
PQoidStatus, 769
PQoidValue, 769
PQoptions, 753
PQparameterStatus, 753
PQparamtype, 768
PQpass, 752
PQping, 745
PQpingParams, 744
PQport, 752
PQprepare, 759
PQprint, 768
PQprotocolVersion, 754
PQputCopyData, 782
PQputCopyEnd, 782
PQputline, 784
PQputnbytes, 785
PQregisterEventProc, 794
PQrequestCancel, 779
PQreset, 744
PQresetPoll, 744
PQresetStart, 744
PQresStatus, 762
PQresultAlloc, 789
PQresultErrorField, 762
PQresultErrorMessage, 762
PQresultInstanceData, 794
PQresultSetInstanceData, 794
PQresultStatus, 761
PQresultVerboseErrorMessage, 762
PQsendDescribePortal, 775
PQsendDescribePrepared, 774
PQsendPrepare, 774
PQsendQuery, 773
PQsendQueryParams, 773
PQsendQueryPrepared, 774
PQserverVersion, 754
PQsetClientEncoding, 785
PQsetdb, 740
PQsetdbLogin, 740
PQsetErrorContextVisibility, 786
PQsetErrorVerbosity, 786
PQsetInstanceData, 794
PQsetnonblocking, 776
PQsetNoticeProcessor, 790
PQsetNoticeReceiver, 790
PQsetResultAttrs, 789
PQsetSingleRowMode, 777
PQsetvalue, 789
PQsocket, 755
PQsslAttribute, 755
PQsslAttributeNames, 756
PQsslInUse, 755
PQsslStruct, 756
PQstatus, 753
PQtrace, 786
PQtransactionStatus, 753
PQtty, 753
PQunescapeBytea, 772
PQuntrace, 787
PQuser, 752
PREPARE, 1636
PREPARE TRANSACTION, 1639
ps
  мониторинг активности, 668
psql, 4, 1792
Python, 1196

Q

querytree, 253, 390
quote_ident, 195
  в PL/Perl, 1189
  использование в PL/pgSQL, 1124
quote_literal, 195
  в PL/Perl, 1189
  использование в PL/pgSQL, 1124
quote_nullable, 196
  в PL/Perl, 1189
  использование в PL/pgSQL, 1124

R

radians, 187
radius, 246
RADIUS, 579
RAISE
  в PL/pgSQL, 1145
random, 188
rank, 298
  гипотезирующая функция, 297
read committed, 415
real, 122
REASSIGN OWNED, 1641
record, 179
recovery.conf, 664
REFRESH MATERIALIZED VIEW, 1642
regclass, 178
regconfig, 178
regdictionary, 178
regexp_match, 196, 210
regexp_matches, 197, 210
regexp_replace, 197, 210
regexp_split_to_array, 197, 210
regexp_split_to_table, 197, 210
regoper, 178
regoperator, 178
regproc, 178
regprocedure, 178
regr_avgx, 293
regr_avgy, 293
regr_count, 293
regr_intercept, 293
regr_r2, 293
regr_slope, 294
regr_sxx, 294
regr_sxy, 294
regr_syy, 294
regtype, 178
REINDEX, 1644
reindexdb, 1831
RELEASE SAVEPOINT, 1647
repeat, 198
repeatable read, 417
replace, 198
RESET, 1648
RESTRICT
  с DROP, 87
  действие внешнего ключа, 55
RETURN NEXT
  в PL/pgSQL, 1128
RETURN QUERY
  в PL/pgSQL, 1128
RETURNING, 91
RETURNING INTO
  в PL/pgSQL, 1121
reverse, 198
REVOKE, 60, 1649
right, 198
ROLLBACK, 1653
rollback
  psql, 1822
ROLLBACK PREPARED, 1654
ROLLBACK TO SAVEPOINT, 1655
ROLLUP, 104
round, 187
ROW, 42
row_number, 298
row_security_active, 314
row_to_json, 272
rpad, 198
rtrim, 198

S

SAVEPOINT, 1657
scale, 187
SCRAM, 573
SECURITY LABEL, 1659
sec_to_gc, 2439
seg, 2556
SELECT, 8, 93, 1661
  определение типа результата, 361
  список выборки, 107
SELECT INTO, 1681
  в PL/pgSQL, 1121
sepgsql, 2559
sequence
  и тип serial, 123
serial, 123
serial2, 123
serial4, 123
serial8, 123
serializable, 418
session_user, 310
SET, 327, 1683
SET CONSTRAINTS, 1686
SET ROLE, 1688
SET SESSION AUTHORIZATION, 1690
SET TRANSACTION, 1692
SET XML OPTION, 549
setseed, 188
setval, 279
setweight, 253, 389
  назначение веса определённым лексемам, 253
set_bit, 207

set_byte, 207
set_config, 327
set_limit, 2535
set_masklen, 250
shared_ispell, 2566
shared_ispell_dicts, 2567
shared_ispell_mem_available, 2567
shared_ispell_mem_used, 2567
shared_ispell_reset, 2567
shared_ispell_stoplists, 2567
shared_preload_libraries, 1022
shobj_description, 322
SHOW, 327, 1695, 2010
show_limit, 2534
show_trgm, 2534
SIGHUP, 496, 569, 572
SIGINT, 484
sign, 187
SIGQUIT, 484
SIGTERM, 484
SIMILAR TO, 209
similarity, 2534
sin, 188
sind, 188
single-user mode, 1878
skeys, 2448
sleep, 243
slice, 2449
smallint, 120
smallserial, 123
snowball-ext, 2568
Solaris
  настройка IPC, 479
  разделяемая библиотека, 1012
  скрипт запуска, 473
SOME, 292, 300, 303
sort, 2455
sort_asc, 2456
sort_desc, 2456
soundex, 2444
SP-GiST (см. индекс)
SPI, 1212
  примеры, 2397
SPI_connect, 1213
SPI_copytuple, 1257
spi_cursor_close
  в PL/Perl, 1187
SPI_cursor_close, 1240
SPI_cursor_fetch, 1236
SPI_cursor_find, 1235
SPI_cursor_move, 1237
SPI_cursor_open, 1231
SPI_cursor_open_with_args, 1232
SPI_cursor_open_with_paramlist, 1234
SPI_exec, 1218
SPI_execp, 1230
SPI_execute, 1215
SPI_execute_plan, 1228
SPI_execute_plan_with_paramlist, 1229
SPI_execute_with_args, 1219
spi_exec_prepared
  в PL/Perl, 1187
spi_exec_query
  в PL/Perl, 1186
spi_fetchrow
  в PL/Perl, 1187
SPI_finish, 1214
SPI_fname, 1246
SPI_fnumber, 1247
spi_freeplan
  в PL/Perl, 1187
SPI_freeplan, 1263
SPI_freetuple, 1261
SPI_freetuptable, 1262
SPI_getargcount, 1225
SPI_getargtypeid, 1226
SPI_getbinval, 1249
SPI_getnspname, 1253
SPI_getrelname, 1252
SPI_gettype, 1250
SPI_gettypeid, 1251
SPI_getvalue, 1248
SPI_is_cursor_plan, 1227
SPI_keepplan, 1241
spi_lastoid
  в PL/Tcl, 1175
SPI_modifytuple, 1259
SPI_palloc, 1254
SPI_pfree, 1256
spi_prepare
  в PL/Perl, 1187
SPI_prepare, 1221
SPI_prepare_cursor, 1223
SPI_prepare_params, 1224
spi_query
  в PL/Perl, 1187
spi_query_prepared
  в PL/Perl, 1187
SPI_register_relation, 1243
SPI_register_trigger_data, 1245
SPI_repalloc, 1255
SPI_returntuple, 1258
SPI_saveplan, 1242
SPI_scroll_cursor_fetch, 1238
SPI_scroll_cursor_move, 1239
SPI_unregister_relation, 1244
split_part, 198
SQL/CLI, 2200
SQL/Foundation, 2200
SQL/Framework, 2200
SQL/JRT, 2200
SQL/MED, 2200
SQL/OLB, 2200
SQL/PSM, 2200
SQL/Schemata, 2200
SQL/XML, 2200

ограничения и совместимость, 2234
sqrt, 187
sr_plan, 2568
ssh, 492
SSI, 414
SSL, 489, 800
в libpq, 757
с libpq, 750
sslinfo, 2571
ssl_cipher, 2571
ssl_client_cert_present, 2571
ssl_client_dn, 2571
ssl_client_dn_field, 2572
ssl_client_serial, 2571
ssl_extension_info, 2572
ssl_issuer_dn, 2571
ssl_issuer_field, 2572
ssl_is_used, 2571
ssl_version, 2571
SSPI, 575
STABLE, 1000
START TRANSACTION, 1697
START_REPLICATION, 2012
statement_timestamp, 235
stddev, 294
stddev_pop, 294
stddev_samp, 294
STONITH, 640
string_agg, 292
string_to_array, 285
strip, 253, 389
strpos, 198
subarray, 2456
subltree, 2470
subpath, 2470
substr, 199
substring, 190, 205, 209, 210
sum, 292
suppress_redundant_updates_trigger, 346
svals, 2448
systemd, 472
RemoveIPC, 480

T

tablefunc, 2572
tableoid, 57
tan, 189
tand, 189
Tcl, 1171
tcn, 2581
template0, 591
template1, 591, 591
test_decoding, 2582
text, 125, 250
text2ltree, 2471
tid, 178
time, 129, 131
time span, 129
time with time zone, 129, 131
time without time zone, 129, 131
TIMELINE_HISTORY, 2010
timeofday, 235
timestamp, 129, 132
timestamp with time zone, 129, 132
timestamp without time zone, 129, 132
timestamptz, 129
TOAST, 2130
в сравнении с большими объектами, 817
и пользовательские типы, 1033
режимы хранения столбцов, 1346
to_ascii, 199
to_attname, 2437
to_attnum, 2437
to_atttype, 2438
to_char, 224
и локали, 598
to_date, 224
to_hex, 199
to_json, 272
to_jsonb, 272
to_namespace, 2438
to_number, 224
to_regclass, 316
to_regnamespace, 316
to_regoper, 316
to_regoperator, 316
to_regproc, 316
to_regprocedure, 316
to_regrole, 316
to_regtype, 316
to_schema_qualified_operator, 2437
to_schema_qualified_relation, 2437
to_schema_qualified_type, 2437
to_timestamp, 224, 235
to_tsquery, 253, 384
to_tsvector, 253, 383
transaction_timestamp, 235
translate, 199
trigger, 179, 1064
triggered_change_notification, 2581
trim, 190, 205
true, 139
trunc, 187, 251, 251
TRUNCATE, 1698
tsm_handler, 179
tsm_system_rows, 2583
tsm_system_time, 2583
tsquery (тип данных), 148
tsquery_phrase, 254, 390
tsvector (тип данных), 147
tsvector_to_array, 255
tsvector_update_trigger, 255
tsvector_update_trigger_column, 255
ts_debug, 255, 406
ts_delete, 253
ts_filter, 254

ts_headline, 254, 387
ts_lexize, 256, 408
ts_parse, 256, 407
ts_rank, 254, 386
ts_rank_cd, 254, 386
ts_rewrite, 254, 391
ts_stat, 256, 393
ts_token_type, 256, 408
tuple_data_split, 2477
txid_current, 323
txid_current_if_assigned, 323
txid_current_snapshot, 323
txid_snapshot_xip, 323
txid_snapshot_xmax, 323
txid_snapshot_xmin, 323
txid_status, 323
txid_visible_in_snapshot, 323

U

UESCAPE, 24, 27
unaccent, 2584, 2585
UNION, 109
  определение типа результата, 359
uniq, 2456
Unix-сокет, 747
unknown, 179
UNLISTEN, 1700
unnest, 285
  для tsvector, 255
UPDATE, 13, 90, 1701
  RETURNING, 91
upper, 190, 289
  и локали, 597
upper_inc, 289
upper_inf, 289
UPSERT, 1618
URI, 745
UUID, 149
uuid-ossp, 2586
uuid_generate_v1, 2586
uuid_generate_v1mc, 2586
uuid_generate_v3, 2586

V

VACUUM, 1705
vacuumdb, 1834
vacuumlo, 2660
VALUES, 111, 1708
  определение типа результата, 359
varchar, 125
var_pop, 294
var_samp, 295
VM (см. Карта видимости)
void, 179
VOLATILE, 1000
VPATH, 1063

W

WAITLSN, 1711
WAL, 722
WHERE, 101
WHILE
  в PL/pgSQL, 1133
width, 246
width_bucket, 187
WITH
  внутри SELECT, 112, 1661
WITH CHECK OPTION, 1537
WITHIN GROUP, 35
word_similarity, 2534

X

xid, 178
xmax, 57
xmin, 57
XML, 150
XML option, 151, 549
XML-функции, 257
xml2, 2587
xmlagg, 260, 292
xmlcomment, 257
xmlconcat, 257
xmlelement, 258
XMLEXISTS, 261
xmlforest, 259
xmlparse, 150
xmlpi, 259
xmlroot, 260
xmlserialize, 150
xmltable, 263
xml_is_well_formed, 261
xml_is_well_formed_content, 261
xml_is_well_formed_document, 261
XPath, 262
xpath_exists, 263
xpath_table, 2589
xslt_process, 2591